The OMAC API Open Architecture Methodology

Abstract

Open modular architecture controllers technology offers great potential for integration of process improvements and better satisfaction of process requirements. With an open architecture, controllers can be built from best value components from best in class services. The need for open-architecture controllers is high, but vendors are slow to respond. One reason for the delay in industry action is that no clear open-architecture solution has evolved. In an effort to promote open architecture control solutions, the Technologies Enabling Agile Manufacturing (OMAC) program for Intelligent Closed Loop Processing (ICLP) sponsors an Application Programming Interface (API) workgroup. This paper reviews the OMAC API workgroup's open architecture efforts. The goal of the OMAC API workgroup is to specify standard APIs for a set of open architecture controller components. The review includes the OMAC API definition of open architecture, as well as the advantages and impediments to open architectures. An overview of the OMAC API reference model including the OMAC API core modules, application framework and specification methodology is given. Overview of the OMAC API specification methodology includes discussion of the API definition and language strategy, definition of the client/server behavior model, methods for transparency of distributed communication, and benefits of reusability of software components through foundation classes.

1   BACKGROUND

Most Computer Numerical Control (CNC) motion and discrete control applications incur high cross-vendor integration costs and vendor-specific training. On the other hand, in a modular, standard-based, open-architecture controller modules can be added, replaced, reconfigured, or extended based on the functionality and performance required. Modifications to a module should provide equivalent or better functionality as well as offer different performance levels. Ideally, the module interfaces should be vendor-neutral, plug-compatible and platform independent.

However, it is important to note that openness alone does not achieve plug-and-play. One vendor's idea of openness need not be the same as another vendor's. Openness is but one step towards plug-and-play. In reality, plug-and-play openness is dependent on a standard. This leads to the following definition of an open architecture controller:

An open architecture control system is defined and qualified by its ability to satisfy the following requirements:

Open provides ability to piece together systems from components, ability to modify the way a controller performs certain actions, and ability to start small and upgrade as a system grows.

Modular refers to the ability of controls users and system integrators to purchase and replace controller modules without unduly affecting the rest of the controller, or requiring extended integration engineering effort.

Extensible refers to the ability of sophisticated users and third parties to incrementally add functionality to a module without completely replacing it.

Portable refers to the ease with which a module can run on different platforms.

Scalable allows different performance levels and size based on the platform selection. Scalability means that a controller may be implemented as easily and efficiently by systems integrators on a stand-alone PC, or as a distributed multi-processor system to meet specific application needs.

Maintainable supports robust plant floor operation (maximum uptime), expeditious repair (minimal downtime), and easy maintenance (extensive support from controller suppliers, small spare part inventory, integrated self-diagnostic and help functions.)

Economical allows the controller of manufacturing equipment and systems to achieve low life cycle cost.

Standard Interfaces allow the integration of off-the-shelf hardware and software components and a standard computing environment to build a controller. Standard interfaces are vital to plug-and-play.

Degree of openness can be evaluated by comparing a claim of openness against the above requirements. Herein, the concept of an open-architecture control system that supports openness, and the auxiliary requirements will be identified as *"open, openness or open architecture."*

1.1 Advantages of Open Architecture Technology

Based on specific instances of problems encountered by users of proprietary controllers, the following list containing explicit requirements of an open-architecture was generated. An open architecture should be able to do the following:

- provide a migration path from existing practices;

- allow an integrator/end user to add, replace, and reconfigure modules;

- provide the ability to modify spindle speed and feed rate according to some user-defined process control strategy;

- allow access to the real-time data at a predictable rate up to the servo loop rate;

- allow full 3-D spatial error correction using a user-defined correction strategy;

- decouple user interface software and control software and make control data available for presentation;

- provide communication functions to integrate the controller with other intelligent devices;

- increase the ability for 3rd party software enhancements. Examples of 3rd party enhancements include:

  - replace a PID control law with a more sophisticated Fuzzy Logic control law
  - collect servo response data with a 3rd party tool, and set tuning parameters in the appropriate control law
  - add a force sensor, and modify the feed rate according to a user defined process model
  - perform high resolution straightness correction on any axis
  - replace the user interface with a 3rd party user interface that emulates a user interface familiar to your machine operators.

The initial validation strategy for the OMAC API would be to insure that this list of capabilities can be addressed.

1.2   Impediments to Open Architecture Technology

It is difficult to define a specification that is safe, cost-effective, and supports real-time performance.

A specification must factor in current practices, as well as anticipate evolving technologies. To be successful, the open architecture definition must be implementable with current computer technology and skills. Further, an open architecture specification cannot be so rigidly defined as to precludde future technology upgrades. An open architecture specification must be able to grow.

Of great importance within the controls domain is the requirement for guaranteed, hard-real-time performance. Without this, safety is at risk. Safety is a major concern voiced within the controller industry which is especially concerned with the issues of liability and allocation of responsibility within an open architecture paradigm. New industry practices would have to be adopted for open architecture controllers. A greater responsibility would be placed on the integrator. Conformance testing would play a larger role. Conformance could require regression and boot-up testing and verification procedures to guarantee proper operation.

A further hindrance is the fact that modules are not "self-contained." Defining an infrastructure within which the modules can operate is necessary and quite difficult. We consider the *infrastructure* to be defined as the services that tie the modules together and allow modules to use platform services. The infrastructure is intended to hide specific hardware and platform dependence; however, this is often difficult to achieve.

Containing the scope of the specification is also difficult. Openness goes beyond run-time APIs. There can be "other" APIs, including configuration, integration, and initialization. As an example, consider the simple use of a math library API. Even there, specification of the math library implementation must be done to select either a floating point processor or software emulation.

Finally, group and industry dynamics can be a problem. From a workgroup perspective, getting people to agree can be a challenge because there are difficult trade-offs in modularization, scope-covered,life cycle benefits to be realized, costs, time to market, and complexity. It is recognized that industry will find it difficult to adopt the OMAC paradigm, due to entrenchment in the legacy of prior implementations, the "comfort zone" of past practice and culture, the investment hurdle to effect change, and the shortage of skilled resources. Proper acculturation, training and education of people and an orderly introduction, demonstration, robustization, motivation, and scale-up will be needed to realize the potential benefits. From an industry perspective, many companies do not perceive any direct benefit from an open architecture. Overcoming the workgroup inertia and industry skepticism by promoting and articulating the benefits of open architecture remains a fundamental key to open architecture acceptance.

2   REFERENCE MODEL

The OMAC API requirements were derived from the OMAC or "Open Modular Architecture Controller" requirements document [OMA94]. The OMAC document describes the problem with the current state of controller technology and prescribes open modular architectures as a solution to these problems. OMAC defines an open architecture environment to include Platform, Infrastructure, and Core Modules. OMAC API defines a *module* to be a component of the system that is sufficiently defined such that an integrator can replace it with another component from a third party. This, at a minimum, will maintain the same service through the same interfaces, have the same set of states, and have the same state transition conditions. Interchangeable modules may differ in their performance levels. Modules may provide more functionality (added value) than required in the specification. A controller may have more than one instance of a module [See Scalability].

Using the OMAC specification model as a baseline, Figure 1 diagrams the OMAC API Core Modules including a brief description of a module's general functional requirements. The Core Modules have the following general responsibilities:

Task Coordinator modules are responsible for sequencing operations and coordinating the various motion, sensing, and event-driven control processes. The task coordinator can be considered a finite state machine (FSM) accepting directives one at a time from an operator or as a stored sequence of instructions in the form of a Control Plan.
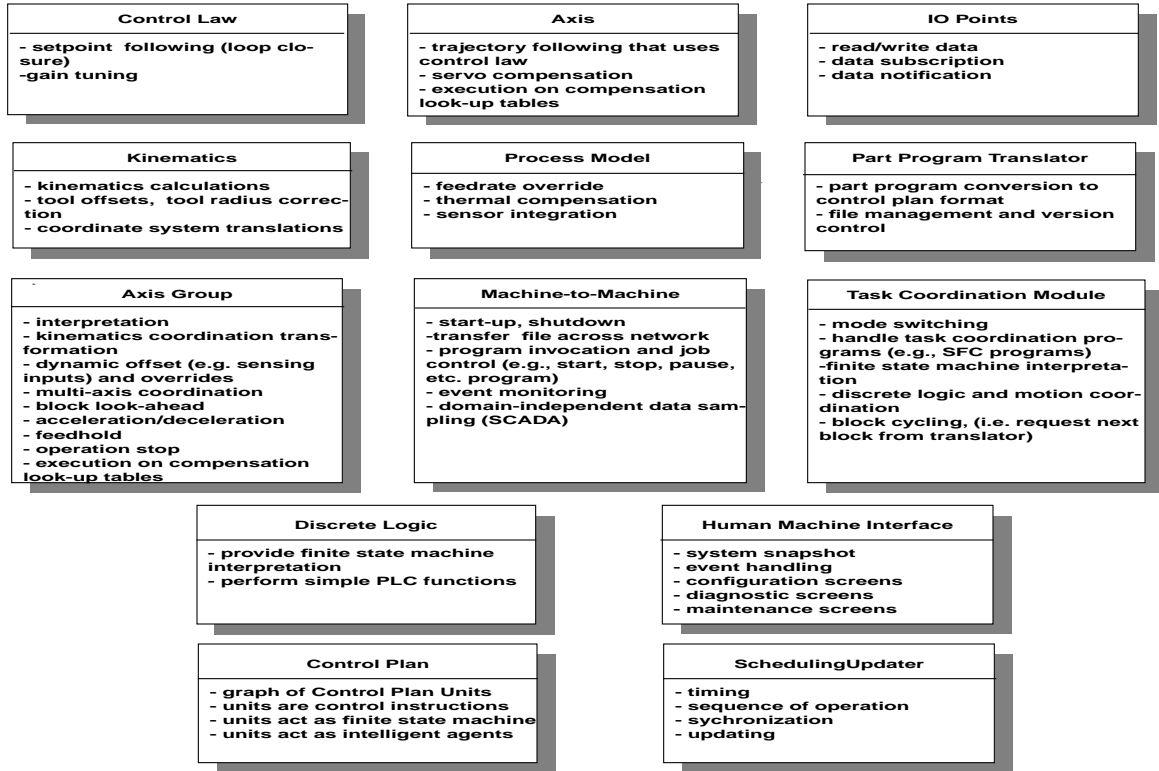
Control Law
- setpoint following (loop closure)
-gain tuning

Axis
- trajectory following that uses control law
- servo compensation
- execution on compensation look-up tables

IO Points
- read/write data
- data subscription
- data notification

Kinematics
- kinematics calculations
- tool offsets, tool radius correction
- coordinate system translations

Process Model
- feedrate override
- thermal compensation
- sensor integration

Part Program Translator
- part program conversion to control plan format
- file management and version control

Axis Group
- interpretation
- kinematics coordination transformation
- dynamic offset (e.g. sensing inputs) and overrides
- multi-axis coordination
- block look-ahead
- acceleration/deceleration
- feedhold
- operation stop
- execution on compensation look-up tables

Machine-to-Machine
- start-up, shutdown
-transfer file across network
- program invocation and job control (e.g., start, stop, pause, etc. program)
- event monitoring
- domain-independent data sampling (SCADA)

Task Coordination Module
- mode switching
- handle task coordination programs (e.g., SFC programs)
-finite state machine interpretation
- discrete logic and motion coordination
- block cycling, (i.e. request next block from translator)

Discrete Logic
- provide finite state machine interpretation
- perform simple PLC functions

Human Machine Interface
- system snapshot
- event handling
- configuration screens
- diagnostic screens
- maintenance screens

Control Plan
- graph of Control Plan Units
- units are control instructions
- units act as finite state machine
- units act as intelligent agents

SchedulingUpdater
- timing
- sequence of operation
- sychronization
- updating

Figure 1: OMAC API Core Modules

Part Program Translator modules are responsible for translating the part program into a Control Plan. Similarly, translations for IEC 1131-3[IEC93] programs and other formats are responsible for producing Control Plans.

Axis Group modules are responsible for coordinating the motions of individual axes, transforming an incoming motion segment specification into a sequence of equi-time-spaced setpoints for the coordinated axes.

Axis modules are responsible for servo control of axis motion, transforming incoming motion setpoints into setpoints for the corresponding actuators.

Kinematics Models modules are responsible for kinematic transformations including geometric correction, tool offsets, and effects of tool wear. Computing forward and inverse kinematics, mapping and translating between different coordinate systems, and resolving redundant kinematic solutions are examples of kinematic model functionality.

Control Law components are responsible for servo control loop calculations to reach the specified setpoints.

Human Machine Interface or HMI modules are responsible for remotely handling data, command, and event service of an internal controller module. Defining a presentation style (e.g., GUI look and feel, or pendant keyboard) is not part of OMAC API effort.

Process Model is a component that contains dynamic data models to be integrated with the control system. Process control components not specified in this architecture produces adjustments or corrections to nominal rates and path geometry in the form of this component. Feedrate override and thermal compensation are examples of process model functionality. The process model is important to the concept of extensible open systems.

4

Discrete Logic modules are responsible for implementing discrete control logic or rules that can be characterized by a Boolean function from input and internal state variables to output and internal state variables. More than one discrete logic module is permitted, but not necessary. Multiple discrete logic modules is similar to having many PLC's networked together within the same computing platform.

I/O Points are responsible for the reading of input devices and writing of output devices through a generic read/write interface. The goal is to provide an abstraction for the device driver. Logically related IO may be clustered within a disrete logic module.

Scheduling Updater is a module that provides centralized scheduling functionality, that includes, timing, synchronization and sequencing. This mechanism is provided since most real-time operating system do not explicitly provide timed updating sequencing.

Control Plan is an aggregation of classes that form the basis of control and data flow within the system. A Control Plan Unit is a base class that contains instructions for a module. A Control Plan consists of a graph of Control Plan Units. *Motion Segments* are a derived class of Control Plan Units for motion control. *Discrete Logic Units* is a derived class of Control Plan Units for discrete logic control.

Machine-to-Machine modules are responsible for connecting and communicating to controllers across different domains (address spaces). An example of this functionality is the communication from a Shop Floor controller to an individual machine controller on the floor.

2.1 Reference Architecture

In the interest of flexibility, scalability, and reusability, OMAC API does not specify a fixed architecture. Instead, OMAC API specifies API for components to support the OMAC core modules. At a higher level, the assembly of the OMAC modules into a system requires an integration architecture and an assembly strategy described below for connecting modules. Suggestions are offered, but are not mandated.

OMAC API assumes a module assembly described by this abstraction hierarchy:

- Foundation Classes

- Framework Components

- Core Modules

- Integration Architecture

- Application Architecture

The foundation classes are the building blocks that may be found in multiple modules. For example, the class definition of a point would be found in most modules. Framework components are instances of foundation classes that can be integrated into the core modules. For example, LinearPath or CircularPath objects are framework components of a Control Plan Unit for motion. The core modules have the functionality as previously outlined. An integration architecture describes a configuration methodology for component topology, timing, and inter-component communication protocols. An application archesitecture specifies components and interconnections selected for a particular application, from the choices allowed by the generic or reference integration architecture. With the application architecture, users can develop and run programs. Some candidate distributed reference architectures include the following: agent-based, DCOM [DCO], CORBA [COR91], RCS [Alb91], OSACA [OSA96], or EMC [PM93].

2.2 Application Framework

In the OMAC API, an application control systems is built as a set of connected modules that use other module services through the published API. The OMAC API specifies module APIs aimed for the system integrator. At this level, the system integrator links ".o" object files (or linked libraries) to assemble a
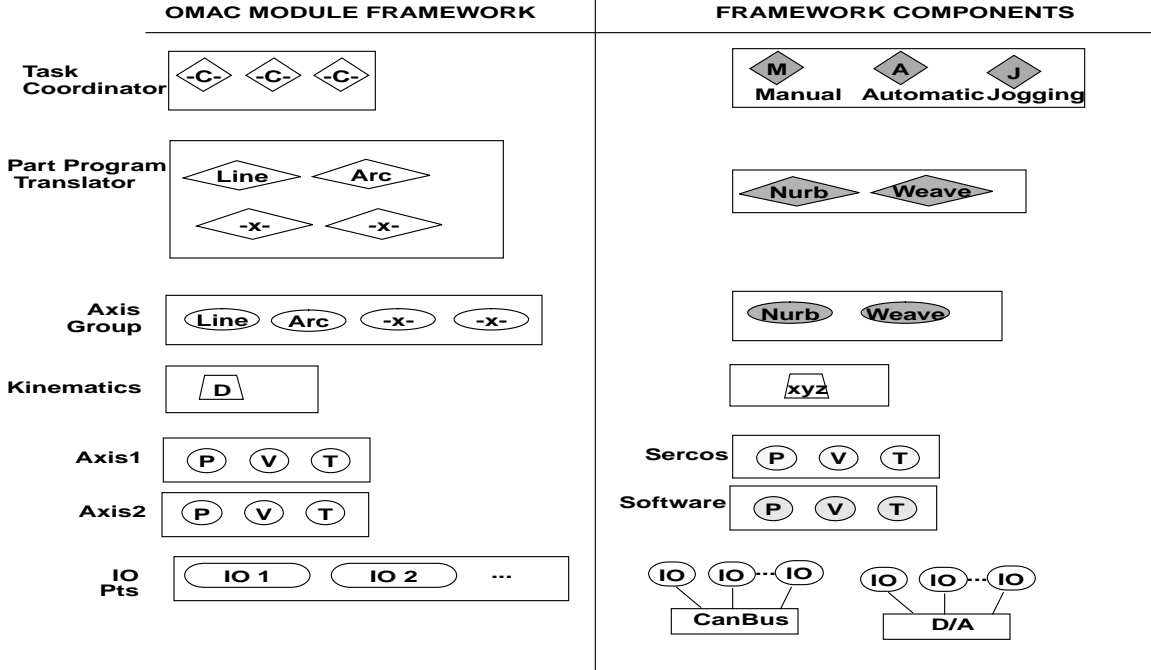
Figure 2: Control Framework

controller. The .o's correspond to procured modules bought as commercial off-the-shelf technology (COTS). The assembly of OMAC API modules in such a manner is referred to as the *framework paradigm*.

Object-oriented *frameworks* are sets of prefabricated software and building blocks that are extensible and can be integrated to execute well-defined sets of computing behavior. Frameworks are not simply collections of classes. Rather, frameworks come with rich functionality and strong "pre-wired" interconnections between the object classes.

This contrasts with the procedural approach where there is difficulty extending and specializing functionality; difficulty in factoring out common functionality; difficulty in reusing functionality that results in duplication of effort; and difficulty in maintaining the non-encapsulated functionality. With frameworks, application developers do not have to start over each time. Instead, frameworks are built from a collection of objects, so both the design and the code of a framework may be reused.

In the OMAC API framework the prefabricated building blocks are the COTS implementations of the OMAC modules and framework components. As a simple example, Figure 2 illustrates a framework for a typical controller application. An application developer buys the modules, and then the application developer "puts the pieces together."

Within the example, there is a task coordinator module which has containers for inserting capabilities (in the figure represented by a -C- framed by a diamond). The capabilities include Manual, Automatic or Jogging. The application developer is free to put one or more of these capabilities into the task coordinator or develop a unique capability. For Part Program Translation and Axis Group, the application developer is already provided line and arc path descriptions but can plug in Nurb (Non-Uniform Rational B-Spline) or Weave path descriptions. Once again, application developers could uniquely develop a path description. For the Axis modules, the application developer has the possibility to do position (P), velocity (V) or torque (T) control in software, hardware or some combination of hardware and software. For software P control, the application developer would select a control law object from the Software set. For hardware P control, the application developer would select a control law object from the Sercos set.

Using the OMAC API framework paradigm, application development involves three groups:

Users define the behavior requirements and the available resources. Resources include such items as hard-

6

Figure 3: OMAC API Example System

ware, control and manufacturing devices, and computing platforms. For behavior, the user defines the performance and functionality expected of the controller. Performance includes such characteristics as how fast or how accurate the application must be. Functionality defines the controller capability such as the ability to handle planar part features versus complex part features.

System Integrators select modules and framework components to match the application performance and functional requirements. The system integrator configures the modules to match the application specification. The system integrator uses an integration architecture to connect the selected modules and verifies the system operation. The system integrator also checks compliance of modules to validate the user-specification of performance and timing requirements.

Control Component Vendors provide module and framework component products and support. For control vendors to conform to an open architecture specification, they would be required to conform to several specifications including the following:

- customer specifications
- module class specification
- system service specification

The system service describes the platform and infrastructure support (such as communication mechanisms) and the resources (disks, extra memory, among others) available. Computer boards have a device profile that includes CPU type, CPU characteristics and the CPU performance characteristics. Included within the profile is the operating system support for the CPU. A spec sheet or computing profile [SOS94] is required to describe the system service specification that would include such areas as platform capability, control devices, and support software.

2.3 Application Example

Figure 3 illustrates the major systems of a controller as they might be configured for a typical numerical control application. The application example describes programmed numerical control for a two-axis lathe. Axis components are assumed be the same for each axis and consist of a PWM motor drive, an amplifier enable control, an amplifier fault status signal, an A-QUAD-B encoder with marker pulse and switches for home and axis limits. Spindle drive components are assumed to provide a facility for setting spindle speed and direction and to start and stop spindle rotation. A machine sensor system is assumed to consist of a set of analog and digital sensors monitoring coolant temperature and oil pressure. A machine safety system is assumed to consist of a set of input switches monitoring E-Stop, Power-Up and Reset. A control pendant is assumed to provide an operator with a simple set of control functions including part program selection, Cycle Start/Stop, Feedhold, Feedrate Override, Manual Data Input and Manual Jogging. Machine part programs are assumed to be in EIA RS274D format. Control pendant is assumed to display machine status to an operator including indication, machine modes, program status and machine diagnostics.

Figure 3 shows the implementation as two sets of components, the larger box of components is the real-time controller and the smaller box contains the HMI mirror. More on the mirrored HMI system will be presented later. The example controller is made of seven major systems. Each system is made up of one or more replaceable modules. Modules are tied together through exposed interfaces. A key concept in modular open architectures is that the system may be incrementally adapted to changing requirements. Three mechanisms for adapting the system are adding modules, replacing modules and reconfiguring modules by reconnecting the interfaces. The seven major systems that make up the example controller include:

IO consists of one or more IO points. Each IO point corresponds to a sensor or actuator. IO points can be grouped that would correspond to some logical grouping of device io points. The IO system entire and provides a naming service to return a handle to an IO point or IO group. The IO module interfaces are used by axis modules, axis group modules, logic control modules and human interface modules.

Axis System consists of one or more axis modules which, in turn, contain control law modules. Each axis module requires two or more IO module interfaces. These represent sensor input and actuator output. Each axis module provides a command interface that is normally connected to an axis group module. Control Law modules may provide additional interfaces that allow features such as status information for the human interface, monitoring/tuning of internal parameters, real time data collection and real time algorithm modification. Each axis module references one or more control law modules which apply a servo algorithm to generate a new actuator command based on current sensor readings, commanded set points and machine state.

Trajectory Generation System consists of one or more axis group modules, a process module and a kinematics module. An axis group module requires at least one control loop interface for each coordinated degree of freedom in the computed trajectory. It may also require additional control interfaces if it supports algorithmically related motions (electronic gearing). An axis group module may also require one or more IO module interfaces to provide sensor modified generation such as impedance control. An axis group requires a process model module and a kinematic model module to handle application and device modeling. An axis group module provides at an interfaces which is normally connected to a task coordinator module.

Process Model provides support to each axis group to receive dynamic input such as feedrate and spindle override.

Kinematics Model provides support to an axis group for coordinate transformation information, such as relative offset.

Task Coordinator System normally consists of one task coordinator module. A task coordinator is the central point for coordination of actions. A task coordinator understands the controller configuration to say what modules are in the system and how to start up the modules. A task coordinator is the controller's main sequencing engine, "what happens when," or the highest level Finite State Machine

within the controller. A task coordinator may provide an interface normally used by the human interface module for machine mode and program sequencing.

Discrete Logic System consists of one or more discrete logic modules. Each of these discrete logic modules is a finite state machine, similar in functionality to the task coordinator module. The system normally contains a large number of discrete logic modules with a variety of requirements for IO module interfaces. Logic control modules provide an interface to the task coordinator module that allow status and event transition information to be conveyed. Logic control modules may also provide an interface to be used by part program interpreter modules. Discrete logic module operation is distinguished from control law module operation by the fact that logic control modules execute Boolean equations to generate new IO output values and detect event transitions based on IO inputs and machine state.

Part Program Translator System consists of one or more part program translator modules. Part Program Translator is responsible for reading and translating programs which represent machine operation and tooling. Part Program Translator output is a list of control plans to be interpreted by the task coordinator, motion subsystem or discrete logic subsystem. A part program interpreter uses several system infrastructure services - primarily file system services. A part program interpreter provides an interface that is normally connected to a human interface module.

Human Machine Interface system is composed of a set of HMI objects which mirrors the state of the controller objects. The main assumption is that HMI objects are a snapshot of control system objects and use proxy agents for communication.

3 SPECIFICATION METHODOLOGY

To satisfy the OMAC open architecture specification, a standard API for each of the Core Modules would be defined. Consequently, the primary goal of the OMAC API workgroup is to define standard API for the Core Modules. This section will refine the concept of "API" and describe the OMAC API specification methodology. The API specification methodology applies the following principles:

- Stay at API level of specification. Use IDL to define interfaces.

- Do not specify an infrastructure.

- Use Object Oriented technology.

- Use general Client Server model, but use state-graph to model state behavior.

- Use Proxy Agents to hide distributed communication.

- Finite State Machine (FSM) is model for data and control.

- Define Foundation Classes to foster the concept of reusable assets.

- Mirror system objects in human machine interface.

The following sections will discuss these principles.

3.1 API Specification

API stands for Application Programming Interface, and refers to the programming front-end to a conceptual black box. The math function "`double cos(x)`" specifies the function name, calling sequence, and return parameter, not how the cosine is implemented, be it table lookup or Taylor series. Of importance to the API specification is the function "signature" and its calling and return sequence, assuming of course, that cosine doesn't take too long. Behavior is an explicit element within the API definition and relies on a defined state transition model. A (standard) API is helpful because programming complexity is reduced when one

TESTBEDS

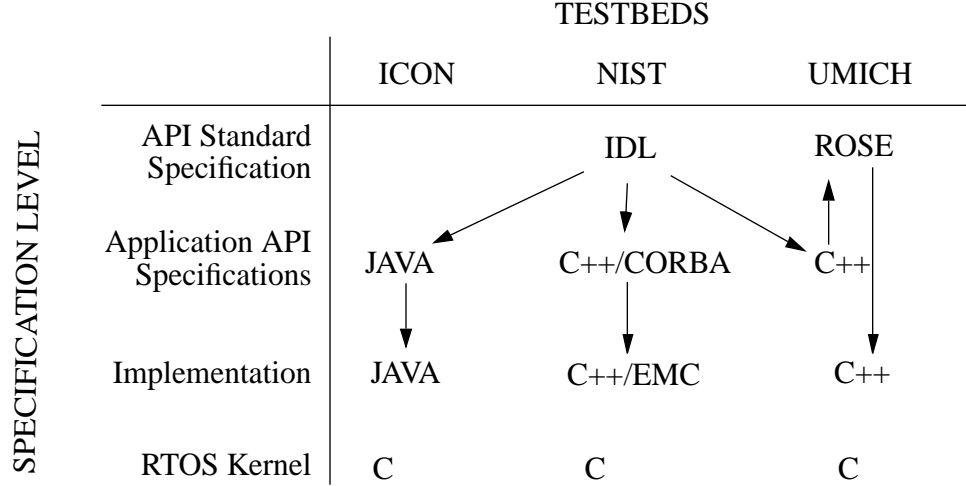|	ICON	NIST	UMICH
SPECIFICATION LEVEL			
API Standard Specification		IDL	ROSE
Application API Specifications	JAVA	C++/CORBA	C++
Implementation	JAVA	C++/EMC	C++
RTOS Kernel	C	C	C

Figure 4: Specification Language Mapping

alternative exists as opposed to several. For example, the cosine signature is generally accepted as `cos(x)`, not `cosine(x)`. This is a small but significant standardization.

At a programmatic level, the importance of a standard API can be seen within the Next Generation Inspection Project (NGIS) at NIST[NGI]. The NGIS project has integrated three commercial sensors and one generic sensor into the Coordinate Measuring Machine controller. Taming diversity was a problem. Each sensor had a different "front-end" - one had a Dynamically Linked Library (.DLL) interface, one had a memory mapped interface, one had a combination port and memory mapping. None of the sensors had the same API. Yet, all of the sensors were "open."

There exists a problem selecting the API specification language. The specification language must be flexible enough to support a variety of implementation languages and platforms. OMAC API chose IDL, or the Interface Definition Language, for its specification language [COR91]. IDL is a technology-independent syntax for describing interfaces. In IDL, interfaces have attributes (data) and operation signatures (methods). IDL supports most object-oriented concepts including inheritance. IDL translates to object-oriented (such as C++ and JAVA) as well as non-object-oriented languages (such as C). IDL specifications are compiled into header files and stub programs for direct use by application developers. The mapping from IDL to any programming language could potentially be supported, with mappings to C, C++ and JAVA available.

To clarify the problem of unifying the specification, consider the mapping of the OMAC API IDL onto three different validation testbeds. Figure 4 illustrates mapping IDL to the different implementation strategies. For ICON, the standard API in IDL has to be mapped into JAVA. At the University of Michigan, they are using the ROSE CASE tool to design their controller. ROSE accepts C++ header through a reverse engineering process. At the NIST testbed, the IDL will be translated into C++ headers and use the Enhanced Machine Controller and its infrastructure[PM93]. For these three implementations, only the IDL specification can be mapped into all the languages needed to support the applications.

3.2   Object Oriented Technology

OMAC API uses an object-oriented (OO) approach to specify the modules' API with class definitions. The following terms will define key object-oriented concepts. A *class* is defined as an abstract description of the data and behavior of a collection of similar objects. Class definitions aggregate both data and methods to offer *encapsulation*. An *object* is defined as an instantiation of a class. For example, the class `SERCOS-Driven Axis` describes objects in the running machine controller. A 3-axis mill would have three instantiations of that class − the three objects described by that class. An *object-oriented program* is considered a collection of objects interacting through a set of published APIs. A by-product of an object-oriented approach is *data abstraction* which is an effective technique for extending base types to meet the programmer needs.

A "complex number" data abstraction, for example, is certainly more convenient than manipulating two doubles.

3.2.1 Inheritance

Inheritance is useful for augmenting data abstraction. OO classes can inherit the data and methods of another class through class derivation. The original class is known as the *base or supertype class* and the class derivation is known as a *derived or subtype class*. The derived class can add to or customize the features of the class to produce either a specialization or an augmentation of the base class type, or simply to reuse the implementation of the base class. To achieve a framework strategy, all OMAC API class signatures (methods) are considered "virtual functions." Virtual functions allow derived classes to provide alternative versions for a base class method.

Using an Axis module as a server, assume that all the axis does is set a variable x.

```
class Axis
{
  virtual void set_x(float x);
private:
  double myx;
}

application()
{
  Axis ax1;
  ax1.set_x(10.0);
}
```

To extend the server, a base class to add an offset to its value before each set is derived. This could also be achieved on the server side if so desired.

```
class myAxis : public Axis
{
    virtual void set_x(float x){ x= x + offset; Axis::set_x(x); }
  private:
    double myx;
    double offset; // set elsewhere for offset calculation
}

application()
{
    Axis ax1;
    myAxis ax2;
    double val;
    double offset;

    val=10.0;
    ax1.set_x(val+offset); // explicit offset in application code
    ax2.set_x(val);        // offset hidden by configuration
}
```
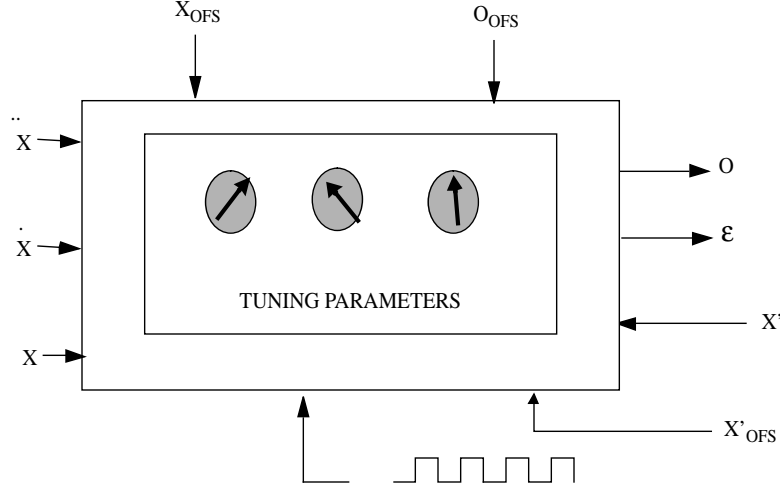
Figure 5: General Control Law

3.2.2 Specialization

OMAC API leverages the OO concept of inheritance to use base and derived classes to add specialization. When defining a control law, one has many options including PID, then Fuzzy, Neural Nets, and Nonlinear. This plethora of options implies a need to contain the realm of possibilities. The OMAC API approach is to define a base type (generally corresponding to one of the OMAC Core Modules) and then add specialized classes.

The control law module illustrates the base and derived class specialization. The responsibility of the Control Law module is conceptually simple – use closed loop control to cause a measured feedback variable to track a commanded setpoint value using an actuator.

Figure 5 illustrates the definition of a base control law. The concept of tuning is encapsulated within the black box and is conceptually controlled via "knob turning." The concept of accepting third party signal injection is handled by the inclusion of pre-and post-offsets (or injection points). These offsets allow sensors or other process-related functionality to "tap" and dynamically modify behavior by applying some coordinate space transformation. The IDL definition of the illustrated control law module follows. Each IDL `attribute` maps into a get and a set accessor methods.

```
interface CONTROL_LAW{
    // Attributes
    attribute double X;
    attribute double Xdot;
    attribute double Xdotdot;
    attribute double output;
    attribute double actual;
    attribute double following_error;
    attribute double XOffset, OutputOffset, XprimeOffset;
    // Operations
    API::Status calculate_control_cmd();
    API::Status init();    // clear time history
};
```

Each `CONTROL_LAW` specialization is a subtype whereby each subtype inherits the definition of the super-type. By applying this concept, an evolutionary process evolves to adapt to changes in the technology. At
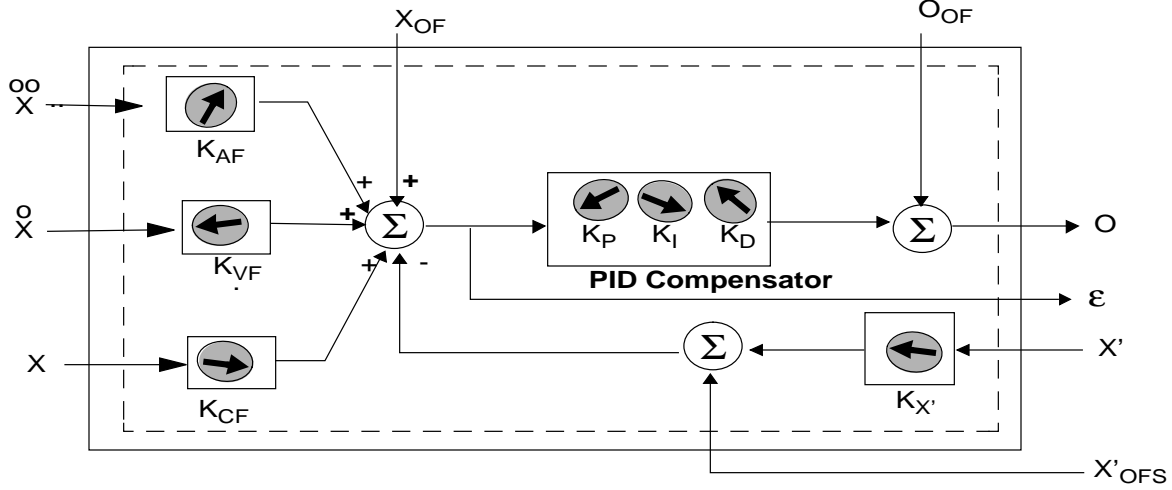
12

Figure 6: PID Control Law

first, only highly-demanded subtypes, such as PID, were handled. Figure 6 conceptually illustrates the PID specialization of the control law. The IDL definition of the PID control law follows.

```
interface PID_TUNING: CONTROL_LAW{
    // Attributes
    attribute double Kp, Ki, Kd;
    attribute double Kaf, Kvf, Kcf, Kxprime;
};
```

OMAC API also uses inheritance to maintain levels of complexity. Level 0 would constitute base functionality seen in current practice. Level 2 would constitute functionality expected of advanced practices. Level 3, 4,..., n would constitute advanced capability seen in emerging technology, but unnecessary for simple applications.

3.3 Client Server Behavior Model

OMAC API adopts a client server model of inter-module communication. In the client/server model, a module is a *server* and a user of a module is called a *client*. Modules can act as both a client and a server and cooperate by having clients issue requests to the servers. The server responds to client requests. A client invokes *class methods* to achieve behavior. A client uses *accessor methods* to manipulate data. Accessor methods hide the data physical implementation from the abstract data representation. The server reacts to the method invocation and performs the corresponding method implementation and sends a reply (either an answer or a status) back to the client.

As a server, a module services requests from clients that can be immediately satisfied or that may require multiple cycles. Multiple cycle service requests require state space logic to coordinate the interaction. OMAC API define three types of service requests: (1) parametric requests, (2) commanded requests, and (3) administrative methods.

Parametric service requests are generally the get/set methods and are, in theory, immediately satisfied. They do not require state space logic.

Command service requests are command methods which may run one or many axis cycles - such as move_to absolute position. Repeated cycles of the same command methods require a state transition mechanism for coordination between the client and the server. Service requests require a state space to coordinate the client server interaction.
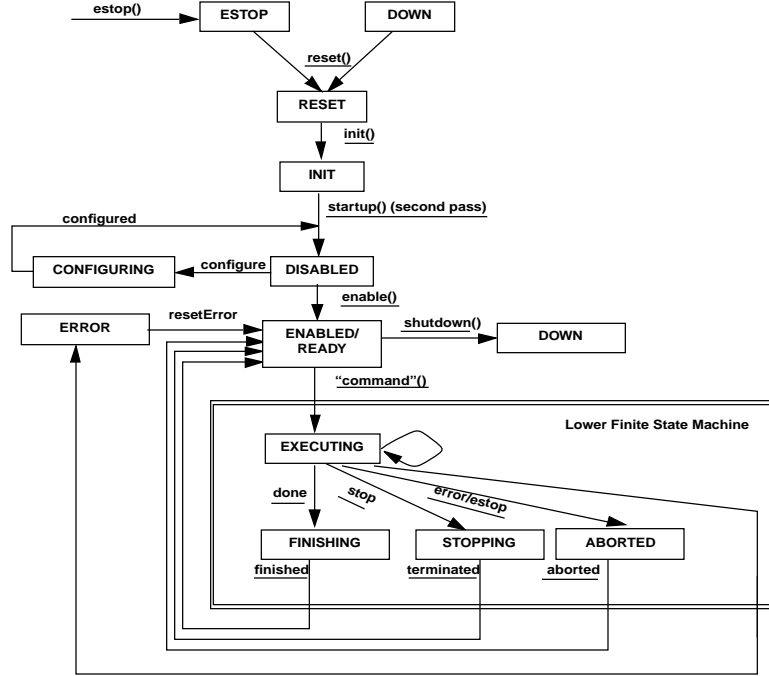
Figure 7: Generalized State Diagram

Administrative service requests coordinate the execution of a module, for example, `processServoLoop,` `enable, disable` for Axis module. The `processServoLoop` function provides cyclic execution - e.g., axis module is executed once per servo loop period. In this mode, the axis software would be running as a data flow machine: at every period, it accesses the data (e.g., commanded position, actual feedback) and derives a new setpoint. Administrative methods can require a state space, such as enabled/disabled/running, but will be considered as part of the service request state space.

Service requests that run multiple cycles require a state space mechanism for coordination between the client and the server. Without such state coordination, the client could not monitor the server's progress toward satisfying the client's request. OMAC API adopts a generalized state model as illustrated in Figure 7. A state model describes the behavior of a module and consists of states and state transitions. For clients to understand a server module's control logic and react accordingly, the client and server must agree on the same state graph representation of valid states and state transitions. Figure 7 shows the typical states found in any control module – start, initialized, configured, enabled/ready, executing, and aborted.

For purposes of representing a module's state space, the concept of administrative states and process commanded states are combined within the state graph. Most of the enumerated states are administrative in order to coordinate the module computational engine. To service a command request, the module enters into the "executing" state. In the "executing" state, client/server coordination uses a lower finite state machine for coordination. This lower finite state machine for command services is module dependent.

3.3.1 Threads of Control

Parametric and Command service methods may be separated from Administrative methods and executed in separate threads of control. Figure 8 illustrates a server with multiple clients and two processes: an Axis Group process for issuing setpoints and an integration architecture process to coordinate execution. Generally, the commanded service requests would come from an Axis Group module that is issuing setpoints to multiple axes. Another thread of execution will handle module integration by sequencing execution of the axis module. This integration module may be tied to some hardware device (such as a timer) to guarantee periodic execution behavior.
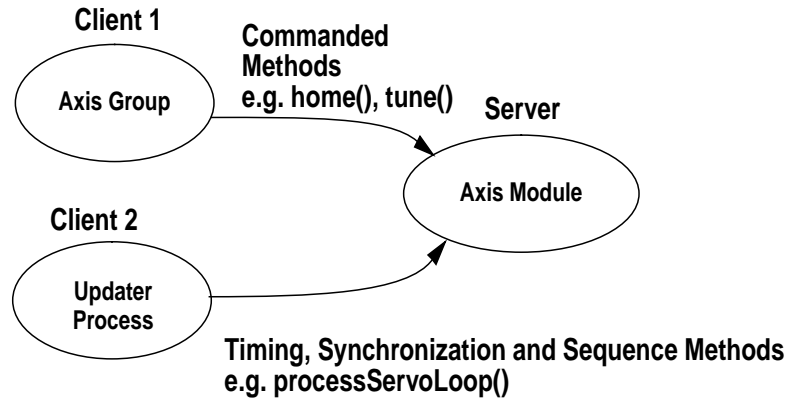
14

Figure 8: Multiple Threads of Control

The following code snippet illustrates the "Small Picture" in assembling the modules. An integration architecture reads and writes between external interfaces between modules, and then in a separate thread of execution calls each modules administrative "execute function."

The example will develop a connection between an Axis Group module, an Axis module and an actuator and encoder IO points. First, the object naming and registry will be sketched. The integration creates object references (i.e., io1, io2, ax1, trajgen1) and then binds addresses to the objects through some name registration.

```
integration_process_init(){
    // initialize parameters
    IOpoint  io1= new IOpoint("encoder1");
    IOpoint  io2= new IOpoint("actuator1");
    Axis ax1= Axis("Axis1");
}
```

In this case, IO points were created and then an Axis and AxisGroup constructor was called. This information would typically be furnished by the Task Coordinator module because it is defined to have the knowledge of the application configuration.

Next, an integration process to synchronize the execution of the modules follows.

```
// Integration architecture puts this together
integration_process(){

    // Use state to cause module to execute - probably at different rates
    if(ax1.state() == running) ax1.processServoLoop();
    if(axgrp1.state() == running) axgrp.execute();
    ...
}
```

The Axis module ax1 has a method processServoLoop which is cyclical process that inputs, computes and outputs. This process could also be a finite state machine that depend on the state of the ax1 object.

```
Axis::processServoLoop(){
    Measure value;
```

```
// Read sensor - i.e, the current encoder value with IO system
value=io1.get();

// Set the next actuator value
ax1.set(value);

// Get the next value set by the trajectory generator
value=get_command();

// Put out value to DAC, (scaling done by io system)
value=ax1.get_output();
io2.put(&value);
}
```

3.4   Proxy Agent Technology

Client/server interaction can be local or distributed. In **local** interaction, the client uses a class definition to declare an object. When a client accesses data or invokes object methods, interaction is via a direct function call to the corresponding server class member. At its simplest, local interaction can be achieved with the server implemented as a class object file or library. Interaction is connected by binding the client object to a newly created server object implementation. Such a binding could be done by static linking, or with a dynamically linked library (DLL) or through a register and bind process that does not use the linker symbol table.

When **distributed** service is needed a *proxy agent* is used which is a set of objects that are used to allow the crossing of address-space or communication domain boundaries[M.S86]. The class describing a proxy agent uses the API of some other class (for which it is a proxy) but provides a transparent mechanism that implements that API while crossing a domain boundary. The proxy agent could use any number of lower level communication mechanisms including a network, shared memory, message queues, or serial lines.

Below is a code example to illustrate the concept of proxy agents. We will assume that we have defined an axis module by the class Axis that has but one method `set_x();`. The following code would be found in the axis module header file (or API specification):

```
class Axis : Environment
{
public:
   void set_x();
private:
   double myX;
}
```

As a user, one would develop code to connect or bind to the axis module server, which in this case has the name "Axis1." The `_bind` service is similar to a constructor method, but returns a server reference pointer and keeps track of the number of client pointer references to the server. The bind establishes a client/server relationship with the axis module. The application code is the client, and when Axis methods are invoked, a message is sent to the server. In the following code, the application sets the x variable to 10.0:

```
application(){
    Axis * a1;
    a1 = Axis::_bind("Axis1");
    a1→set_x(10.0);
}
```

If the server is colocated with the application, it is trivial to implement the object server. The `Axis::set_x` implements the value store.

```
Axis::set_x(double _x){ myX = _x; }
```

However, for distributed communication, `Axis::set_x` is defined twice - once on the client side and once on the server side. On the client side we set up the remote communication, which in this case, is a sketch of a remote procedure call.

```
Axis::set_x(double _x){
    callrpc(host, prognum, versnum, procnum, inproc, in, outproc, out)
}
```

On the server side, a server waits for service events (such as the `bind`, and the `set_x` method). A corresponding `Axis::set_x` is defined to handle the x variable store. The server technology could handle events in the background or use explicit event handling. In either case, the server actions are transparent to the client.

```
Axis::set_x(double _x){ myX = _x; }

server(){
    /* register rpc server name */
    while(1) { /* service events */ }
}
```

Given the proxy agent fundamentals, the next step is to adapt this to FSM control. Below is a code sketch of an `Axis` class that defines two methods `process_servo_loop` and `home`. An important aspect of the `Axis` implementation is to make the proxy agent *transparent*. To be transparent, a class must define methods that support local or remote method invocation identically. In order to achieve this, an FSM class is defined and when the `home` method is invoked, it inserts a `HOME_EVENT` event into the `Axis` FSM. The FSM has an internal queue for handling events. The FSM may spawn a separate thread of control for event handling.

```
class Axis
{
    FSM AxisFSM;

    process_servo_loop() { AxisFSM.handle_event(PROCESS_SERVO_LOOP_EVENT); }
    home() { AxisFSM.handle_event(HOME_EVENT); }
};

class FSM {
    msg_queue evq;
    int cur_state;

    handle_event(EV_num)
    {
        evq.send(EV_NO);
     }

    FSM_thread() // optional thread, this could be done in handle_event
```

```
        {
            evq.receive(&ev_no);
            call_action(ev_no, cur_state);
        }

        home_update_action() { /* enable homing control plan unit */ }
        process_servo_loop_action() { /* evaluate state */ }
    };
```

Within OMAC API, in order to achieve transparency across implementations, all methods contain a parameter field to allow customization of the infrastructure by defining an environment variable at the end of the parameter list. This is an implicit augmentation performed by an IDL compiler. For any OMAC API calling parameter list, the ENVIRONMENT parameter appears at the end of the calling sequence, as in:

```
    void move(double x, double y, double z, ENVIRONMENT env = default);
```

The ENVIRONMENT can be used in several ways to tailor the infrastructure, such as to specify the remote communication protocol and the necessary parameters during transmission. The ENVIRONMENT can also be used to set an invocation time-out value or to pass security information. The ENVIRONMENT can be a stubbed dummy and ignored by the called method.

The goal of the ENVIRONMENT parameter is to provide transparency between invoking function calls locally or invoking function calls remotely. To provide for transparency between local and remote calls, the ENVIRONMENT parameter field has a default argument initializer so that local (or remote) calls need not supply this parameter.

The actual infrastructure supported by the ENVIRONMENT parameter will not be specified within this OMAC API document. Systems with a proprietary remote communication technology may use the ENVIRONMENT parameter field to enable distributed processing. The ENVIRONMENT can also be used as a trap door to hide other nonstandard operations. To enable compatibility with known remote processing requirements, OMAC API uses accessor functions to manipulate object data members. The data format creates one or two accessor functions − one to set and one to get − as defined by the cases for read only, write only, or read-write combinations.

```
        void set_x(double inx, ENVIRONMENT env=default);
        double get_x(ENVIRONMENT env=default);
```

Note that the ENVIRONMENT parameter at the end of the parameter list is necessary.

3.5   Infrastructure

The infrastructure deals primarily with the computing environment including platform services, operating system, and programming tools. Platform services include such items as timers, interrupt handlers, and inter-process communications. The operating system (OS) includes the collection of software and hardware services that control the execution of computer programs and provide such services as resource allocation, job control, device input/output, and file management. Real Time Operating System Extensions can be considered platform services since these extensions are required for semaphoring, and pre-emptive priority scheduling, as well as local, distributed, and networked interprocess communication. Programming tools include compilers, linkers, and debuggers.

The OMAC API does not specify an infrastructure because many of the infrastructure issues are outside the controller domain and would be better handled by the domain experts. Further, it is more cost-effective to leverage industry efforts rather than to reinvent these technologies. For example, commercial implementations of proxy agent technology are available. Microsoft has developed and released DCOM (Distributed

Common Object Model) for Windows 95 and Windows NT. Many implementations of CORBA (Common Object Request Broker Architecture) are available and Netscape incorporates an Internet Interoperable ORB Protocol (IIOP) inside its browser. The question concerning the hard-real-time capability of such products remains. But, industry is acting to solve this problem. In the interim, control standards that could provide a real-time infrastructure are available [OSA96].

Because there are so many competing infrastructure technologies, OMAC API has chosen to allow the market to decide the course of the infrastructure definition. As such, to achieve plug-and-play module interchangeability, a commitment to a *Platform + Operating System + Compiler + Loader + Infrastructure suite* is necessary for it to be possible to swap object modules.

3.6   Behavior Model

For the OMAC API, *behavior* in the controller is embodied by finite state machines (FSM). OMAC API uses state terminology from IEC1131[IEC93]. An FSM *step* represents a situation in which the behavior, with respect to inputs and outputs, follows a set of rules defined by the associated *actions* of the step. A step is either *active* or *inactive*. *Action* is a step a user takes to complete a task which may invoke one or more functions, but need not invoke any. A *transition* represents the *condition* whereby control passes from one or more steps preceding the transition to one or more successor steps. Zero or more actions shall be associated with each step.

For OMAC API, the finite state machine (FSM) is the principal element of both the data flow and control flow. As outlined previously, the client/server model has command requests. The FSM data are passed within command methods from the sending OMAC API module to the receiving OMAC API module to effect behavior. FSM are then used within a module to handle the control flow. A module executes an FSM until it ends or is superceded by another FSM. How the FSM are implemented is not important to the OMAC API. Rather, a general model of FSM behavior is defined in the API so that a OMAC API module calls FSM state transition methods to initiate state changes.

Passing FSM between modules is fundamental to a flexible, extensible controller. Without an FSM, there is no explicit mechanism for sequencing an RS274 block containing simultaneous instructions. Without the FSM, intelligence must be hard-coded into the Task Coordinator to "understand" in what order to sequence and synchronize the block operations. Internally hard-coding the block decoding within the Task Coordinator prevents easy controller modification or extension.
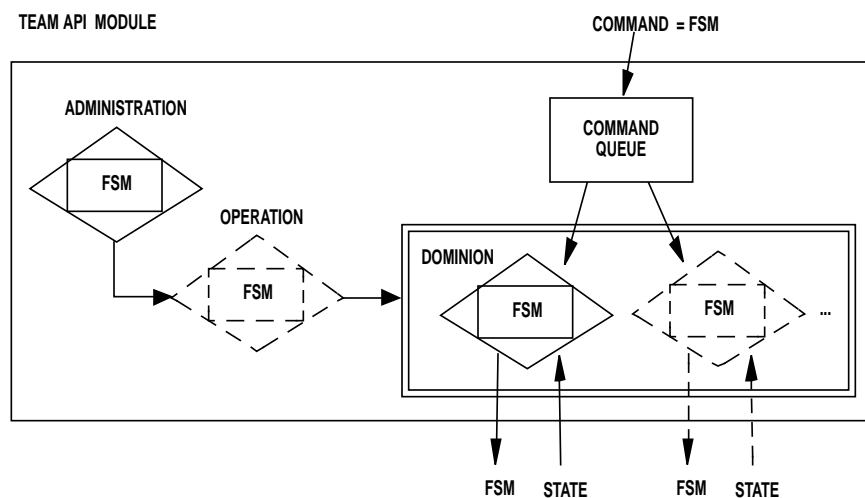

Figure 9: Computation Paradigm

Figure 9 illustrates the different computational elements of a OMAC API module. The OMAC API module supports module system *administration* (`reset, init, startup, shutdown`) that is handled by an administrative FSM. One or more lower level *operation* FSM are possible depending on system complexity.

The OMAC API module contains a queue, possibly of length 1, for queuing commands. Commands are in the form of FSM. The OMAC API module may have one or more FSM executing on a dominion FSM list. The *dominion* FSM list contains FSM that "rule" over other objects. In the diagram, the FSM are represented by a rectangle within a diamond. The dotted line indicates an optional FSM.

For a OMAC API module, there can be several levels of FSM applicability. OMAC API does not dictate the levels of FSM. In general, an outer FSM exists to handle module system administration. Module system administration activities can include initialization, startup, shutdown, and, if relevant, power enabling. The system administrative FSM must follow established safety standards. When the administrative FSM is in the READY state, it is possible to descend into a lower level operation FSM. The operation FSM is optional (as indicated by the dashes in the figure), but is necessary in the task coordinator module. The task coordinator operation FSM is called a Capability. Different Capability are used to handle the different machine modes (manual, auto). When the operation FSM is in the READY state, it too can descend into a lower programming or *dominion* level FSM. The dominion FSM "rules" over other objects by invoking administrative and command methods. Since the module could "rule" over several objects, the potential for multiple dominion FSM exists.

Within any of the three nested levels of FSM mentioned above, there may be more nested levels. For example, at the operation level for part programming, there may be another level of FSM to handle a family of parts. When a particular part is specified, it may invoke a nested FSM that specifies processing to be performed specific to that part. The designer of a particular control system determines the number of nested FSM levels, depending upon the complexity and organization of the controlled system.

A module comes up executing the administrative FSM and after several steps of initialization and insuring safety of operation, a module is READY. At this point, a module is capable of "stepping down" the FSM hierarchy to the dominion FSM. Clients can still invoke administrative methods but can now also invoke command methods to queue an FSM. When enabled, a module will transfer FSM from the queue onto the dominion list. Executing the FSM will generate output to subordinate servers. Clients can invoke parametric methods to query server status. At any point during the processing of commands, a module administrative state can change which will be reflected in the lower level FSM. For example, instructing the module administration to stop() will result in the administrative FSM and all dominion FSM stopping.

Command methods are defined as list management methods that put FSM objects onto the queue. Below, a Task Coordinator would call the Axis Group ag to append the motion segment ms_homing onto the axis group queue.

```
MotionSegment ms_homing;   // parameters set by the part program translator(PPT)
ag→set_next_motion_segment(ms_homing);
```

To sequence an FSM list, at a minimum, calling the execute() method until the isDone() boolean condition is true, can sequence a FSM from start to finish.

```
interface axis_homing : ControlPlanUnit
{
    MotionSegment ms_homing;                // setup by PPT
    AxisGroup ag;                           // set by PPT - discussed later
    ControlPlanUnit handle;
    execute()                               // called by Task Coordinator
    {
      if(firsttime)
    handle=ag→set_next_motion_segment(ms_homing); // message passing!
      else {
    while(!handle→isActive());   // wait till motion segment is active
    while(!handle→isDone());      // then wait until done
      }
```
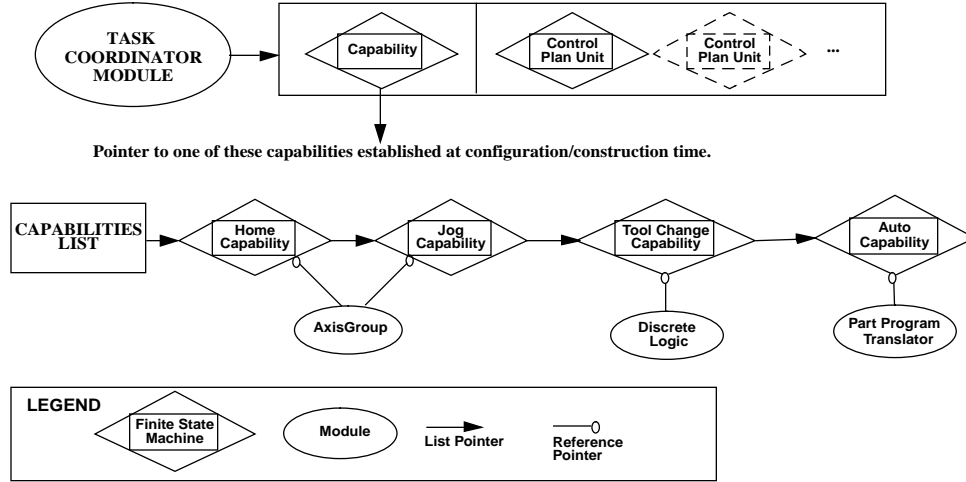
Figure 10: Controller Task Coordinator Capabilities

```
    }

    isDone(){ return(!ag→isHomed()); } // called by  Task  Coordinator
}
```

With this computation paradigm, different OMAC API modules have different command queue and FSM dominion list sizes. The Task Coordinator has a one-element queue as well as a one-element dominion FSM. The Discrete Logic module may have a one element queue, but generally has a multi-item dominion FSM list, some active, some not active, to coordinate the IO points. The Axis Group has a minimum two-element command queue, and generally a one-element dominion FSM list unless some blending of operations or synchronization with a spindle FSM is required. The Axis module only has an extensive administrative FSM. These differences will be further explored.

3.6.1   Task Coordinator

The Task Coordinator has a one-element FSM dominion list. The dominion FSM list is defined by the `Capability` class definition. Associated with the `Capability` FSM is a `ControlPlan` list.

The `Capability` FSM supports `stop`, `start`, `execute`, `isDone` methods. For an application controller, there is list of capabilities that a Task Coordinator can use. Figure 10 illustrates a typical milling CNC application with `Capability` instances. Each `Capability` has reference pointers to OMAC API modules that it uses. Thus, the `Home Capability` and the `Jog Capability` each have reference pointers to the Axis Group. When a `Capability` is executing, it coordinates the servicing of requests from the HMI. When the `Auto Capability` FSM is executing, it interacts with the Part Program Translator.

Figure 11 illustrates a sequence of operations that takes a milling CNC from manual mode to automatic mode. The diagram illustrates that a `Capability` FSM has `start`, `stop`, `execute` methods. There is the assumption that there is a default `Capability`, probably an `Idle Capability`. In the scenario, the operator pushes the `auto` button that causes the HMI to execute the `Manual Capability stop` method, and load the `Auto Capability` onto the Task Coordinator queue. That cycle, the Task Coordinator will see that the `Manual Capability` boolean `isDone` is True and will swap the `Auto Capability` FSM into the dominion FSM list. The operator action to load a program will result in a program name loaded into the Part Program Translator. When the operator pushes the cycle button, it will cause the Auto `Capability` FSM to start sequencing Part Program Translator generated information. Part Program Translator information is called `ControlPlan` and will covered in the next section.

3.6.2 Control Plan Units

When the Task Coordinator dominion is the `Auto Capability`, it coordinates with the Part Program Translator to generate control information. For different applications, the Part Program Translator generates different `ControlPlanUnit` FSM. For the OMAC API, the base type control information is an FSM and is called a `ControlPlanUnit` that may embed other `ControlPlanUnit` FSM. For different control behavior, an FSM has a unique class definition derived from the `ControlPlanUnit`. A series of `ControlPlanUnit(s)` is a `ControlPlan`. A `ControlPlan` can be a simple list to represent sequential behavior or a complex tree to represent parallel controller behavior.

A `ControlPlanUnit` FSM understands how to coordinate and sequence the logic and motion submodules. The `ControlPlanUnit` FSM could put `MotionSegments` on the AxesGroup motion queue. The `ControlPlanUnit` FSM can either put `LogicUnits` on the DiscreteLogic queue or activate `LogicUnits` on the DiscreteLogic dominion list similar to a PLC scanning list. There are also `ControlPlanUnits` for decision making. (e.g., loops, end program and if/then/else). Figure 12 illustrates a `ControlPlan` with one of the `ControlPlanUnits` expanding the hierarchy of possible `ControlPlanUnit` options.

`ControlPlanUnit` has a method `execute_control_plan()` which does not need to be entirely self-contained. It may make use of services of other objects. In addition, the `ControlPlanUnit` acts as a container for embedded `ControlPlanUnits`. These embedded `ControlPlanUnits` are passed to the appropriate server, such as, a `MotionSegment` is passed to the Axes Group module. To use such a sequence of control, the Part Program Translator builds a `ControlPlanUnit` for the Task Coordinator FSM that causes a `MotionSegment` FSM to be pushed onto AxesGroup Queue. It is important to understand that this rippling effect is a fundamental mechanism for passing data through a OMAC API controller. The following section provides a simple code example to illustrate this rippling effect.

3.6.3 Forward Reference

The OMAC API specifies that `ControlPlanUnit` objects allow embedding of dynamic module references and direct method calls. On the surface this approach appears implausible. However, because of proxy agent technology, creating a "forward reference" by dynamically binding to an object is not hard to do. This dynamic binding is beneficial since it eliminates the need for static encoding of methods with id numbers so that methods can execute across domains (address spaces). To enable forward references, the requirement does exist for the infrastructure to support some "`lookup()`" method to map object names to addresses.

As an example, the application of proxy agent technology will be used by Part Program Translator to generate a `ControlPlanUnit` for an `axis_homing` FSM. The `axis_homing` is an FSM with a transition method `execute` and a query method `isDone` to determine FSM completion.

```
interface axis_homing : ControlPlanUnit
```
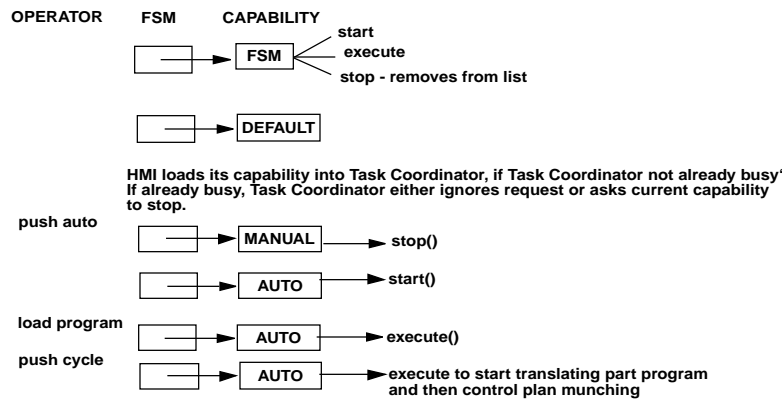

Figure 11: Step Through of a Task Coordinator Capability Sequence
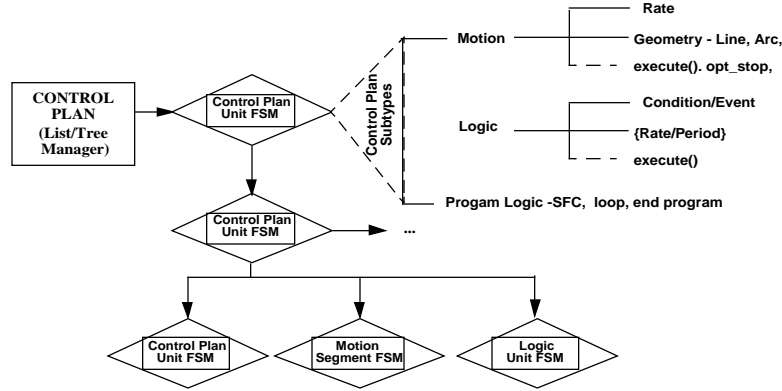
Figure 12: Control Plan Hierarchy

```
{
   attribute MotionSegment ms_homing;   // parameters set by the PPT
     execute()                          // called by Task Coordinator
     {
       if(firsttime)
     ag→set_next_motion_segment(ms_homing); // message passing!
       else if(!ag→isOK());             // do error checking each cycle
     }

   isDone(){ return(!ag→isHomed()); } // called by Task Coordinator
   set_axgrp(char * axgroupname ) { ag=lookup(axgroupname); }
private:
   Axis Group *ag;                      // ag set by the PPT
}
```

The `execute` and `isDone` methods use explicit calls to an Axis Group object. A "forward reference" to the Axis Group object is required. Suppose the Part Program Translator (`PPT`) receives at constructor time the name "*axisgroup1*" for the Axis Group object. Lookup of the "*axisgroup1*" must be available through the underlying proxy agent technology. Without the proxy agent technology, one has to encode the object `ag` and the methods `ag->home` and `ag->isDone`. This extra programming overhead is hidden by the proxy agent technology.

3.6.4 Discrete Logic

The Discrete Logic module is similar to the Task Coordinator module in that it sequences and coordinates actions through dominion FSM. However, for clarity, instead of a monolithic one-element dominion FSM, the Discrete Logic module has a multi-item dominion FSM list. In general, a Discrete Logic dominion FSM could be coded in any of IEC-1131 languages. Figure 13 illustrates the types of FSM that may be found on the Discrete Logic dominion list for a typical milling CNC application. An FSM to handle IO scanning would be expected. An FSM implemented as a Ladder Rung could be expected to handle a relay for turning a Mist pump on. Below one finds a sketch of the activity for turning the IO mist pump on.

```
mist_pump_on_rung()
execute()
{ logic:  trigger relay to turn pump on
          wait till IO/pt says pump is on
```
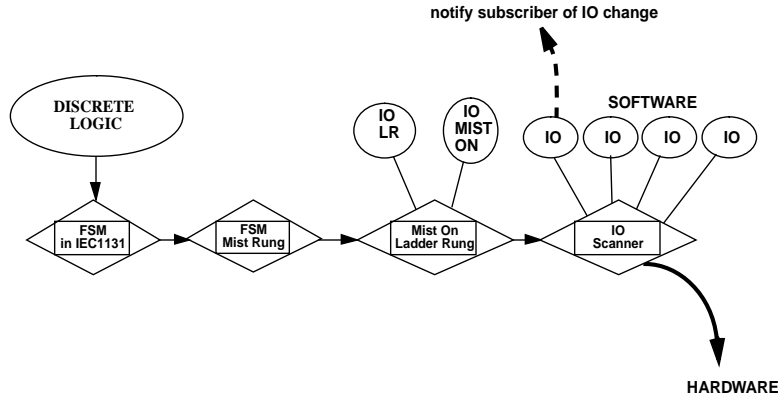
Figure 13: Discrete Logic FSM List

```
            IOmist← on;
    }
```

At a higher level, a hardware-independent Mist FSM would be required to coordinate turning Mist on and off. Below is a sketch of pseudo code to sequence the Mist on operation. For coordination between FSM logic, polling or event-drive alternatives exist to wait for the IO Mist on activity to complete.

```
mist_on_fsm()
{ "MistOn LR IO <- on"  to turn LR=ladder rung on
  "subscribe to event that IO Mist On ==on"
  "wait for event or poll for IO point for Mist On == on "
  "done - deactivate FSM for scanning"
}
```

3.7   Foundation Classes and Data Representation

Exchange of information between modules relies on standard information representation. Such control domain information includes units, measures, data structures, geometry, kinematics, as well as the framework component technology. Figure 14 portrays the conceptual organization of framework component software as defined by foundation classes.

Consider the analogy of building materials. The primitive data types, shown at the bottom of Figure 14, are similar to such raw materials as sand, gravel, and clay. Using foundation classes and aggregating structural components, a control hierarchy of reusable software components can be built. Based upon the reusable foundation classes, these assets can be used to build class libraries for such motion components as sensors, actuators, and pid control laws.

Not all software objects have physical equivalents. Objects such as axis groups are only logical entities. Axis groups hold the knowledge about the axes whose motion is to be coordinated and how that coordination is to be performed. Services of the appropriate axis group are invoked by user-supplied plans (process programs).

OMAC API has chosen two levels of compliance for data definitions. The first level defines named data types to allow type-checking. The OMAC API uses the IDL primitive data types and builds on these data types to develop the foundation classes and framework components. For control domain data modeling, the OMAC API used data representations found in STEP Part Models for geometry and kinematics [Inta, Intb]. Internally, one could, of course, use any desired representation. The STEP data representations were translated from Express into IDL. Representation units are assumed to be in International System of

24

Machining systems/cells; workstations		Plans
Simple machines; tool-changers; work changers		Processes
Axis groups		Fixtures Other tooling
Machine tool axis or robotic joints (translational; rotational)		
Axis components (sensors, actuators)	Control components (pid; filters)	
Geometry (coordinate frame; circle)	Kinematic structure	
Units (meter)	Measures (length)	Containers (matrix)
Primitive Data Types (int,double, etc.)		

Figure 14: Software Reusable Assets

Units, universally abbreviated SI. Below is the basic set of data types which use STEP terminology for data names but reference other terms for clarification.

Primitive Data

- IDL data types include *constants*, *basic data types* (float, double, unsigned long, short, char, boolean, octet, any), *constructed types* (struct, union and enum), *arrays* and *template types* bounded or unbounded sequence and string.
- IEC 1131 types - 64 bit numbers
- bounded string

Time

Length

- Plane angle
- Translation commonly referred to as position
- Roll Pitch Yaw (RPY) commonly referred to as orientation
- STEP notion of a Transform which is composed of a translation + rpy, also commonly referred to as a "pose."
- Coordinate Frame which is defined as a Homogeneous Matrix

Dynamics

- Linear Velocity, Acceleration, Jerk
- Angular Velocity, Acceleration, Jerk
- Force
- Mass

25

- Moment
- Moment of Inertia
- Voltage, Current, Resistance

The second level provides for more data semantics. The OMAC API adopted the following strategy to handle data typing, measurement units, and permissible value ranges. Distinct data representations were defined for specific data types. For example, the following types were defined in IDL to handle linear velocity.

```
// Information Model - for illustrative purposes
typedef Magnitude double;

// Declaration
interface LinearVelocity : Units  {

    Magnitude  value; // should this value be used?
    // Upperbound and Lowerbound, both zero ignore
    Magnitude ub, lb; // which may be ignored

    disabled();
    enabled();
};

// Application
LinearVelocity vel;
```

In this case, linear velocity is a special class. Unit representation is inherited from a general units model. Permissible values are defined as a range from lowerbound to upperbound. The units and range information are optional and may not be used by the application.

Another data typing problem that must be resolved concerns the use of a parameter. Not all parameters are required or need be set by every algorithm. For example, setting the jerk limit may not be necessary for many control algorithms. To resolve the parametric dependency issue it was decided to use a special value to flag a parameter as "not-in-use". This approach seems simpler than having a use_xxx type method for each parameter. For now, OMAC API has decided that setting a parameter to a unrealistic "Not in use Number" (but not actually "Not a Number") value - such as MAXDOUBLE or 1.79769313486231570e+308 - renders a double parameter to be ignored or not-in-use. A similar number would be required for an integer. This works for level 1 and level 2. Within level 2, the methods enable and disable were added to explicitly indicate use of a parameter.

3.8   Human Machine Interface

The primary HMI objective of the OMAC API is to provide the ability to "bolt-on" a Human Machine Interface to the controller. The HMI is intended to be independent of the choice of presentation medium, the dialogue mechanism, the operating system, or the programming language.

OMAC API specifies that every controller object has a corresponding HMI object "mirror". A simplifying assumption is that HMI objects communicate to control objects via proxy agents. Figure 15 illustrates the mirroring of a one axis controller that uses a task coordination module for coordination and sequencing in conjunction with a discrete logic module.

The desired HMI functionality is best understood in the context of simple problems. Three "canonical" problems exist that an HMI module must be able to handle regardless of the interface device. First, the user must be able to receive *solicited information reports* about the state of the controller, such as a current axes position. Second, the user must have *command capabilities* such as set manual mode, select axis, and then
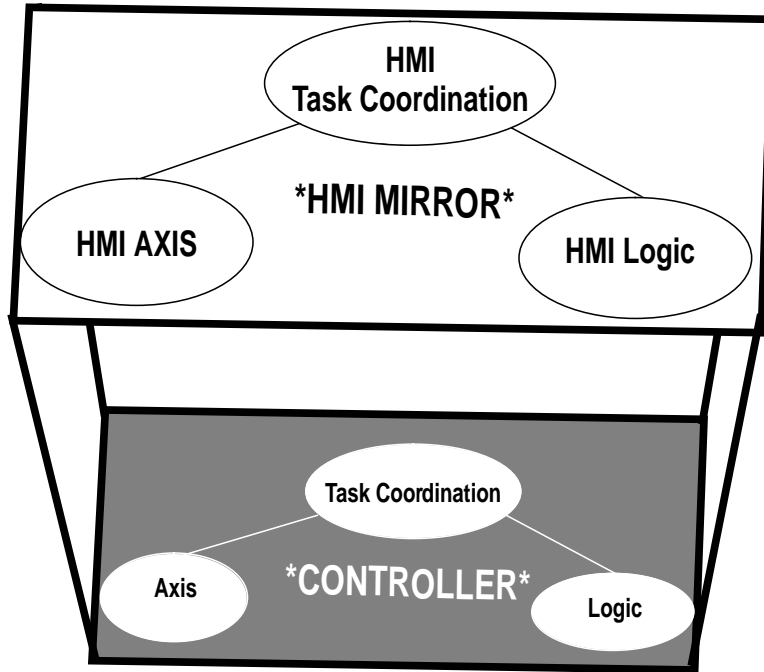
Figure 15: Human Machine Interface Mirrors Controller

jog an axis. Third, the user must be alerted when an exception arises, in other words, handle *unsolicited information reports.* Following is an analysis of how the HMI mirror handles these cases.

To handle the information report functionality, an HMI mirror acts as a remote data base that replicates the state and functionality of the controller object and then adds different presentation views of the object. These HMI mirrors are not exact mirrors of the controller state, but rather contain a "snapshot" of the controller state. Figure 16 illustrates the interaction of the HMI mirror and the control object. In the basic scenario of interaction, the control object is the server and the HMI mirror object is the client. Each HMI mirror uses the accessor functions of get and set to interact with the control object. You will notice that each host controller object and corresponding HMI mirror have a proxy agent to mediate communication.

To handle command functionality, the HMI mirror contains the same methods as the controller object so that a command is issued by invoking a method remotely.

To handle abnormal events when polled monitoring may not be possible, an HMI mirror must serve as a client to the control object so that it can post alert events. For such unsolicited information reports, the control object uses an event notification function, `update_current_view`, in which to notify the HMI mirror that an event has occurred. This notification in turn may be propagated to a higher-authority object.

The following HMI definition gives the method extensions that a control object must support to become a mirrored object.

```
    interface HMI
    {
      // Presentation Methods
      void present_error_view();
      void present_operational_view();
      void present_setup_view();
      void present_maintenance_view();

      // Events - to alert HMI that something has happened
```

27
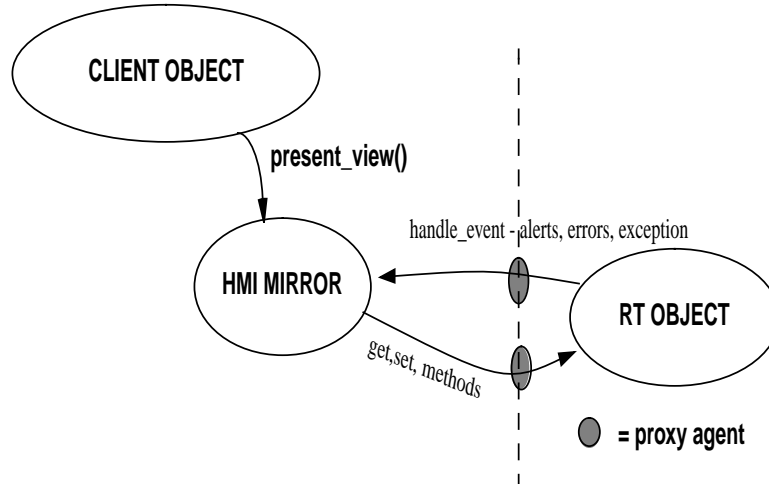
Figure 16: Human Machine Interface

```
    void update_current_view();
};
```

A benefit to using the HMI mirrors is the potential for vendors to supply a control object, as well as a presentation HMI object that can be incorporated into their Operator Interface. As an example of this technology, a tuning package can provide a Windows-based GUI to do some knob turning. Another example, is a tuning package that offers this capability to be plugged inside a Web browser. With this development, unlimited component-based opportunities are available.

4 API

Technical Note: These API are for review and comment only. There is no guarantee of correctness. This specification approximates the intended direction of the final API.

4.1 Disclaimer

This software was produced in part by agencies of the U.S. government, and by statute is not subject to copyright in the United States. Recipients of this software assume all responsibility associated with its operation, modification, maintenance, and subsequent redistribution.

As of April 30, 1997 the IDL Definitions are in state of transition. Some "pure" IDL some hybrid. Eventually there will be no attributes, only get and set methods on these attributes, since IDL does not produce a get/set prefix to the methods. This will not work for non-CORBA-like systems.

4.2 Basic Types

```
1    // All class definitions should register with central name/type server
2    >interface OMAC_CLASS
3    >{
4    >   attribute char * name;
5    >   attribute char * type;
6    >
7    >};
```

28

```
 8
 9    interface OMAC_MODULE
10    {
11      // Administrative State Transition Methods
12      void  estop();
13      void  reset();
14      void  init();
15      void  startup();
16      void  enable();
17      void  disable();
18      void  execute();
19      void  shutdown();
20
21      void  throw_exception();
22      void  resolve_exception();
23
24    >   void  stop();
25    >   void  abort();
26
27
28      boolean isReset();
29      boolean isInited();
30      boolean isEnabled();
31      boolean isDisabled();
32      boolean isReady();
33      boolean isEstopped();
34    >  boolean isException();
35
36    };
37
38    // Level 1 - these will be backed out from the other API definitions
39    //
40    typedef long            API;
41    typedef double          AngularVelocity;
42    typedef Boolean         boolean;
43    typedef Translation     CartesianPoint;
44    interface               CoordinateFrame { /* FIXME */ } ;
45    typedef double          Force;
46    typedef double          Length;
47    typedef double          LinearVelocity;
48    typedef double          LinearAcceleration;
49    typedef double          LinearJerk;
50    typedef double          LinearStiffness;
51    interface               LowerKinematicModel {/* FIXME */ } ;
52    interface               MaintHistory  { /*FIXME*/ };
53    typedef double          Magnatude;
54    typedef double          Mass;
55    typedef double          Measure;
```

```
56    enum                      MOT_OBJ { SPEED, ACCURACY};
57    typedef double            PlaneAngle;
58    interface                 RESOURCE  {/*FIXME*/ } ;
59    interface                 RPY  {/*FIXME*/ } ;
60    interface                 RWCollectable{/*Something equivalent to RogueWave */};
61    typedef                   RWOrdered{/*Something equivalent to RogueWave */};
62    interface                 Time { /* FIXME */ };
63    interface                 Translation  {/*FIXME*/ } ;
64    interface                 UNITS {/*FIXME*/ } ;
65    interface                 UpperKinematicModel {/*FIXME*/ } ;
66
67
68    //?? Or you can assume numbers are flagged not active at
69    //?? construction time.
70    // Below most control parameters would be typed as double
71    #define double_not_active 1.79769313486231570e+308
72    #define long_not_active 0x80000000
73    #define short_not_active 0x8000
74
75
76    // Level 2  Example - not defined here
77
78    interface LinearVelocity : Units  {
79        Magnitude  value; // should this value be used?
80        // Upperbound and Lowerbound, both zero ignore
81        Magnitude ub, lb; // which may be ignored
82        disabled();
83        enabled();
84    };
85    interface Units{ /* FIXME */ };
86
87
```

4.3 Control Plan

```
1    interface ControlPlanUnit
2    { // approximate   a graph structure
3      ControlPlanUnit execute_unit(); // return next ControlPlanUnit
4      // ControlPlanUnit get_next_unit();
5
6      void set_active();     // set when "executing"
7      void set_inactive();
8      boolean isActive();    // for PPT to determin
9
10     // persistence data a la binary image
11     void save(char * file);
12     void restore(char * file);
13
```

```
14      // persistence data in neutral format (pre-configuration)
15      void save_neutral(char * file);
16      void restore_neutral(char * file);
17
18      // The graph is used for non-execution navigatation
19      ControlPlanUnit * cpu[100 /*max*/];
20      attribute int length;   // number of arcs in this graph node
21      attribute int max;      // max number of arc possible should grow dynamically
22      // FIXME: add traversal functions here
23    };
24
25    interface ControlPlan : RWList<ControlPlanUnit>  {};
```

4.4 Scheduling Updater

```
1
2     interface updatable
3     {
4       attribute double period;
5       void update();
6     ]
7
8     interface asych_updater
9     { register_updatable(updatable upd);
10    }
11
12    interface periodic_update : asynch_update
13    {
14        get_timing_interval();
15    }
16
```

4.5 IO

```
1     // Level 1
2     interface IO_PT<T>    // <T>= int, boolean, double, float
3     {
4       <T> get_value();
5       void set_value(<T> v);
6
7       char *get_name();
8       void *set_name(char *name);
9
10
11      attribute (void *) (*monitor) ();
12
13      //?? attribute device-info;    // reference to device info
14
```

```
15    //?? attribute int type;    // 1=read-only, 2=read/write, 0=don't care
16    // or use IO derived type to differentiate types
17    //?? attribute UNITS units;
18    //?? attribute <T> upper_bound;
19    //?? attribute <T> lower_bound;
20    };
21
22    interface callbackNotification
23    {
24      void execute();
25    };
26
27    interface IOPt_Notify
28    {
29      void notify_handlers(); /* list management */
30      void attach(callbackNotification cb);
31    };
32
33    // example derived type
34    // interface IOPt_Nofity_on_sign_change: IOPt_Notify { } ;
35
36    typedef sequence<int> IOvalues;
37    typedef sequence<char *> IOnames;
38    typedef sequence<char *> IOmetadata;
39
40    // Or should this just be an array of IOPts?
41    interface IOgroup
42    {
43      IOvalues get_values();
44      void set_values(IOvalues);
45
46      void add_IO_PT(IO_PT<T> io);
47      IOnames get_names();
48      IOmetadata get_metadata();
49
50    };
51
52    //??typedef sequence<IO_PT<T>> IO_SYSTEM;
53    // A container for a list of IO Points
54    interface IOsystem
55    {
56      IOgroup get_IO_GROUP(char * name);
57      IOPt get_IO_PT(char * name);
58    };
59
60
61    // Example
62    interface myIO : UPDATABLE
```

```
63    {
64      IO_PTshort encoder1;
65      IO_PTshort encoder2;
66      IO_PTlong encoder3;
67
68      void update_now; /* interrupt handler */
69      eventNotification new_sample_available; /* tell clients of new data */
70      set_pacer_clock(divisor);   /* control */
71    };
72
73    // Level 2: Hierarchy of Common IO Points - for type checking
74    // See IO API Document for further details
```

4.6 Task Coordinator

```
 1    // Each capablity is an FSM and types of capabilities include: manual, auto, estop, etc.
 2    // FIXME: What is the relationship of manual to auto and any to estop?
 3    // Internally the capbility is a FSM.
 4    interface Capability
 5    {
 6      void start();
 7      void execute();
 8      void update_cap(); //+ what is difference to execute?
 9      void stop();
10      void abort();
11      void throw_execption();
12      void resolve_execption();
13      boolean isDone();
14      boolean isActive();    //+
15    };
16
17    typedef sequence<Capability> Capabilities;
18
19    // Task Coordinator accepts one capability from a list of capabilities.
20    interface TaskCoordinator : OMAC_MODULE UPDATABLE
21    {
22
23      virtual void update();   //+ inherited from UPDATER
24
25      // Capability List Management
26        void add_to_list(Capability * cap);
27        void remove_from_list(Capability * cap);
28        Capabilities get_list();
29
30      // Current Capability Management
31        Capability * get_current_capability();
32        void set_current_capability(Capability *  cap);
33    };
```

33

4.7 Discrete Logic

```
1    // Discrete Logic Module contains a list of logic units. A PLC like scan
2    // goes down the list and executes each logic unit if it is on. Logic units
3    // will be executed as often as its posted scan rate indicates.
4    // Internally each discrete logic unit is an FSM.
5    // Discrete Logic Units (DLUs) are grouped by scan rates.
6    interface DiscreteLogic : OMAC_MODULE
7    {
8
9      // Logic Units Management
10        DiscreteLogicUnit * create_discrete_logic_unit();
11        void add_logic_unit(DiscreteLogicUnit dlu);
12        void remove_logic_unit(DiscreteLogicUnit dlu);
13        void enable_logic_unit(DiscreteLogicUnit dlu);
14        void disable_logic_unit(DiscreteLogicUnit dlu);
15    };
16
17    // Derived from ControlPlanUnit, see: part program translator
18    interface DiscreteLogicUnit: ControlPlanUnit
19    {
20      attribute integer interval;
21
22      void start();
23      void scan_update();
24      void stop();
25      boolean isOn();
26      boolean turnOn();     // external event causes invokes this method
27      boolean turnOff();
28    };
29
```

4.8 Part Program Translator

```
1    // Level 1 assuming simple File Manipulation
2    interface PPTLevel1
3    {
4        void set_program_name(char * s);
5        char * get_program_name(char * s);
6
7        boolean  checkSyntax();
8        char * get_error_codes(); // or returns file name or file pointer?
9
10       ControlPlan  translate(); // complete translation into ControlPlan
11       ControlPlanUnit * get_next_ControlPlanUnit();  // step by step translation
```

```
12    };
13
14    // Level 2  Production Data Management
15    interface ProductionDataManagement : FILE VERSION
16    {
17      // An OMG standard should be completed by 9/97
18    };
19    interface PPTLevel2
20    {
21        attribute ProductionDataManagement pdm;
22    };
23
24    // Defer interface specification to CAD
25
```

4.9 Axis Group

There are some inconsistencies within the Axis Group module API. The major remaining problem is to resolve the use of the axis group velocity profile generator (VPG) versus having the VGP embedded within a motion segment.

```
1     //+ add accel mode - use instead of enum - windows problem
2     typedef int ACC_MODE;
3     #define  S_CURVE 1
4     #define  TRAPEZOIDAL 2
5
6     interface AxisGroup : OMAC_MODULE UPDATABLE
7     {
8     //+   enum { ERROR, HELD, HOLDING, STOPPED, STOPPING, PAUSED, PAUSING, RE-
SUME, EXECUTING, IDLE };
9
10      // STATE LOGIC
11      // ==============================================
12
13      void hard_stop_axes();  // Stop at max deceleration rate (abort)
14      void pause_axes();       // stop on path
15      void hold_axes();        // stop at end of segment
16      void resume_axes();      // Resumes motion from current point
17
18    //  void     update_axes();
19      void     update();       //+ changed for consistent interface
20
21      int get_current_state();
22      String get_current_state_name();
23      boolean is_OK();
24      boolean is_Executing();
25      boolean is_Held();
26      boolean is_Holding();
27      boolean is_Paused();
```

```
28        boolean is_Pausing();
29        boolean is_Stopping();
30        boolean is_Stopped();
31
32        // These methods could be operator Control Plan Unit
33        void jog_axis( int axis_no, VelocityMeasure speed );
34        void home_axis( int axis_no, VelocityMeasure speed );
35        void move_axis_to(int axis_no, VelocityMeasure speed,  LengthMeasure  to_position);
36        void increment_axis(int axis_no, VelocityMeasure speed, LengthMeasure increment);
37
38        // BUFFERING MANAGEMENT
39        //================================================
40        void set_next_motion_segment( Motion_Segment block);
41        // Motion_Segment get_current_motion_block( );  //hazardous to your controller's health
42        int get_MaxqSize() const;      // largest queue size possible=n
43        void set_qLength(int value); // maximum number of queue members=(1..n)
44        int get_qLength() const;
45        int get_current_qSize();       // number of items in queue=i
46        boolean is_Full();             // number of items = n
47        boolean is_Empty();            // number or items = 0
48
49        void flush();                  // flush all segments
50        void skip();                   // skip to next segment
51        void save_q_context();         // save current queue
52        void restore_q_context();      // restore saved queue
53
54        // FIXME: possibly more queue mgt functions (accessor, query, ... )
55
56        // CONVENIENCE FUNCTIONS TO ACCESS MOTION SEGMENT DATA
57        //================================================
58        length_measure * get_neighborhood() const;
59        LinearVelocity * get_feedrate() const;
60        VelocityMeasure * get_traverseRate();
61        double get_feedrateOverride() const;
62        double get_spindlerateOverride() const;
63        LinearJerk * get_jerkLimit() const;
64        Boolean get_inPosition() const;
65        void set_inPosition(Boolean value); /* private method*/
66
67        // See Note 1
68        Measure get_actual_axis_position( int axis_no );
69        OacVector get_actual_axes_positions( );
70        CoordinateFrame get_xformed_actual_positions();
71        Measure get_commanded_axis_position( int axis_no );
72        OacVector get_commanded_axes_positions( );
73        CoordinateFrame get_xformed_commanded_positions( OacVector axis_positions);
74
75        ACC_MODE get_accMode() const;
```

```
76
77     // KINEMATIC INFORMATION
78     //=================================================
79     // Axis under control
80     CoordinatedAxes * get_coordinatedAxes() const;
81     KinStructure * get_kinStructure() const;
82     void set_kinStructure(KinStructure * value);
83     ToolPartTransforms * get_toolTransform() const;
84     CoordinateFrame * get_baseFrame();
85     void set_baseFrame(CoordinateFrame * value);
86
87     // recovery from fault error, sharing
88     void     inhibit_axis( int axis_no, boolean inhibit );
89     boolean    axis_inhibitd( int axis_no );
90     void     inhibit_spindle( boolean inhibit );
91     boolean     spindle_inhibitd();
92
93     // TRAJECTORY INFORMATION
94     //=================================================
95     void set_blending(boolean flag);       // TRUE=ON, FALSE=OFF
96     void set_single_step(boolean flag);    // TRUE=ON, FALSE=OFF
97
98     // void set_VPG(VelocityProfileGenerator vpg);
99     // VelocityProfileGenerator get_VPG();
100
101    // Timing is now a reference to another object
102    // time_measure get_axisUpdateInterval() const;
103    // void set_axisUpdateInterval(time_measure value);
104    attribute Timing timing;
105
106    void set_physical_limits(Rate * limits); //+ 3-Jun-1997
107    Rate * get_physical_limits();            //+
108    };
109
110    // NOTES
111    // 1. There is a problem in JAVA with returning data type.
112    // Storing into calling parameter as a side effect Side
113    // instead of
114    //        OacVector get_commanded_axes_positions( );
115    // use
116    //        void get_commanded_axes_positions( OacVector positions );
117    // It is possible to redo above in this signature style.
118    // 2. Issue: There are issues as to maximum acceleration of device
119    // versus Control Plan Unit (Motion Segment)
120
121    // Control Plan Class Definitions- Motion Segments
122    #if 0
123    interface CoordinatedAxes
```

```
124     {
125       // Fixme
126     };
127
128     interface OacVecetor
129     {
130       // how does this differ from PathNode
131     };
132
133     interface PathNode
134     {
135       transform get_controltransform();
136       void set_controltransform(transform value);
137     };
138     interface PathElement : public KinematicPath
139     {
140       virtual void initAccDecProfile(LinearVelocity *vel);
141       void set_start_point( PathNode start_point ); // axgroup sets
142       PathNode get_start_point( );
143       PathNode get_end_point();                          // axgroup sets
144       // void set_end_point(PathNode end_point);    // ppt or internal use
145       virtual LengthMeasure get_distance_to_go();
146       boolean isPathComplete();
147       virtual LengthMeasure pathLength();
148       // virtual LengthMeasure pathLength(XYZ xyz); // what is this
149     };
150     #endif
151     interface Rate
152     {
153       void set_nominal_feedrate(double vnom);
154       int set_current_feedrate(double vmax);        // includes override
155       int set_maximum_acceleration(double amax);
156       int set_maximum_jerk(double jmax);
157
158       double get_nominal_feedrate();
159       double get_current_feedrate();                    // includes override
160       double get_maximum_acceleration();
161       double get_maximum_jerk();
162
163       double get_current_velocity();
164       void set_current_velocity(double vcur);
165
166       double get_final_velocity();
167       void set_final_velocity(double vcur);
168
169       double get_current_acceleration();
170       void set_current_acceleration(double acur);
171
```

```
172      int get_acc_state();
173      void set_acc_state(int val);
174      int isDone();
175      int isAccel();
176      int isConst();
177      int isDecel();
178
179      void set_nominal_spindle_speed(double spd); // why here?
180      double get_nominal_spindle_speed();
181    };
182    interface KinematicInfo
183    {
184      void setToolCenter(LengthMeasure& effectiveDisplacement,
185                  CRCMODE cutterRadiusCompensation);
186
187      Xform get_current_frame();
188      void set_current_frame( Xform current_frame );
189
190      KinematicsModule get_kinematics();
191      set_kinematics(KinematicsModule kin);
192    };
193
194    interface VelocityProfileGenerator
195    {
196      AccDecProfile * get_accDecProfile();
197      void set_accDecProfile(AccDecProfile * value);
198
199      void set_blending_point_distance( double distance );
200      double get_blending_point_distance();
201
202      time_measure * get_sampling_time();
203      void set_sampling_time(time_measure * value);
204      /* New   3-Jun-1997 */
205      void hold_segment();
206      void pause_segment();
207      void resume_segment();
208    };
209    // Base Class for Motion Segment
210    // Derived from ControlPlanUnit  - see part program translator
211    interface Motion_Segment  : ControlPlanUnit
212    {
213      attribute KinematicInfo kin;
214
215      void set_vpg(VelocityProfileGenerator * _vpg);
216      VelocityProfileGenerator * get_vpg();
217
218      void set_translational_rate(Rate * rate);
219      Rate * get_translational_rate();
```

```
220
221        void set_orientation_rate(Rate * rate);
222        Rate * get_orientation_rate();
223
224        void set_angular_rate(Rate * rate); // does this belong in axis group?
225        Rate * get_angular_rate();
226
227        // if internal velocity profile generation supply this interface
228        void set_blending_point_distance( double distance );
229        double get_blending_point_distance();
230
231        LengthMeasure calc_distance_remaining(); // axes
232
233        void OacVector get_incremental_distance( );
234        OacVector get_lengths_remaining();   // per axis
235        OacVector calc_next_increment(double feed_override,
236                      double spindle_override,
237                      //? doesn't this need in current_position
238                      double[]    increment = NULL /* ignore side effect */
239                      );
240      boolean start_next_segment(); //? what does this mean init?
241    //?   int init(double cycle_time); //+ 3-Jun-1997
242        void pause_segment();
243        void hold_segment();   /* new */
244        void stop_segment();   /* new 3-Jun-1997 set motion to done */
245        void resume_segment();
246        boolean is_paused();
247        boolean is_held();
248
249        // Program information (file, line number, block) and signals(active)
250        void set_ppb( PartProgramBlock ppb );
251        void segment_started();
252        void segment_finished();
253    };
254    //NOTES:
255    // 1. Handling Termination Condition:
256    // a. Exact Stop = blending distance=0
257
258    #endif
259
```

4.10   Axis

```
1    interface Axis;
2    interface Axis_Absolute_Pos;
3    interface Axis_Acceleration_Servo;
4    interface Axis_Commanded_Output;
5    interface Axis_Dyn;
```

```
6    interface Axis_Error_And_Enable;
7    interface Axis_Force_Servo;
8    interface Axis_Homing;
9    interface Axis_Increment_Pos;
10   interface Axis_Kinematics;
11   interface Axis_Jogging;
12   interface Axis_Limits;
13   interface Axis_Maint;
14   interface Axis_Operation;
15   interface Axis_Positioning_Servo;
16   interface Axis_Rates;
17   interface Axis_Sensed_State;
18   interface Axis_Setup;
19   interface Axis_Velocity_Servo;
20
21   interface Axis
22   {
23     Axis_Absolute_Pos  get_absolute_position(); // bad name...
24     Axis_Acceleration_Servo get_acceleration_servo();
25     Axis_Commanded_Output get_command_output();
26     Axis_Error_And_Enable get_error_and_enable();
27     Axis_Force_Servo  get_force_servo();
28     Axis_Homing get_homing();
29     Axis_Increment_Pos get_increment_position();
30     Axis_Jogging get_jogging();
31     Axis_Positioning_Servo get_positioning_servo();
32     Axis_Sensed_State get_sensed_state();
33     Axis_Velocity_Servo get_velocity_servo();
34
35     enum USE_MODE { OPERATION, SETUP, MAINTENANCE, ENGINEERING};
36     typedef USE_MODE OPER_MODE;
37     attribute USE_MODE usageMode;
38     long processServoLoop( ); // the primary function.
39
40       // State information in hybrid form :(
41     attribute Boolean HOLD;
42     attribute Boolean emergency;
43     long feedHold( );
44     long feedResume( );
45     void estop( );
46     void resetEStop( );
47
48     long checkOTravel( ); //   checked at every servo loop.
49     void calcOpenLoopGain( );
50   };
51   interface Axis_Acceleration_Servo
52   {
53     void start_acceleration_following_action();
```

```
54       void acceleration_update_action();
55       void end_acceleration_following_action();
56       boolean acceleration_following_error();
57     };
58     interface Axis_Commanded_Output
59     {
60       Axis_Position_Cmd  get_position_command();  // returns Position, Velocity, Accel, Force
61       Axis_Velocity_Cmd  get_velocity_command();  // returns Velocity, Accel, Force
62       Axis_Accel_Cmd     get_acceleration_command(); // returns Accel, Force
63       Axis_Force_Cmd     get_force_command();  // returns Force
64
65       void set_position_command( Axis_Position_Cmd  positioning_cmd );
66       void set_velocity_command( Axis_Velocity_Cmd  velocity_cmd );
67       void set_acceleration_command( Axis_Accel_Cmd acceleration_cmd );
68       void set_force_command( Axis_Force_Cmd    force_cmd );
69     };
70     interface Axis_Dyn
71     {
72       attribute Force staticFriction;
73       attribute Force runFriction;
74       attribute Time timeConstant;
75       attribute Length backlash;
76       attribute Length deadband;
77       attribute Mass axmass;
78
79       attribute LinearAcceleration accelerationLimit;
80       attribute LinearAcceleration decelerationLimit;
81       attribute LinearJerk jerkLimit;
82       attribute LinearAcceleration zeroVelAccLim;
83       attribute LinearAcceleration maxVelAccLim;
84
85       attribute Length overshootStepInput;
86       attribute Time risingTimeStepInput;
87       attribute Force quasiStaticLoadLimit;
88       attribute LinearStiffness loadedCaseSpringRate;
89       attribute LinearStiffness worstCaseSpringRate;
90       attribute Mass inertia;
91       attribute Force damping;
92
93     };
94     interface Axis_Error_And_Enable
95     {
96       void reset_axis_action();
97       void disable_axis_action();
98       void enable_axis_action();
99       void e_stop_axis_action();
100    };
101    interface Axis_Force_Servo
```

```
102    {
103      void start_force_following_action();
104      void force_update_action();
105      void end_force_following_action();
106      boolean force_following_error_action();
107    };
108
109    interface Axis_Homing
110    {
111      void start_homing_action(double start_velocity ); // prepares homing
112      void homing_update_action();          // called each servo cycle
113      void stop_homing_action();            // stops homing before completion
114      void homing_complete_action();        // On transition from homing to Enabled
115                                            // – when homing is completed
116      void e_stop_homing_action();          // On transition from homing to E-stopped
117      void    disable_homing_action();      // On transition from homing to disabled
118      boolean is_done();                    // signals when homing is completed
119      boolean is_stopping();
120      boolean homing_error();               // true if error has occurred during homing
121    };
122    interface Axis_Jogging
123    {
124      void start_jogging_action( double target_velocity );
125      void jogging_update_action();
126      void stop_jogging_action();
127      boolean is_done();
128      boolean is_stopping();
129      boolean jogging_error();
130      void e_stop_homing_action();
131      void disable_homing_action();
132    };
133    interface Axis_Kinematics
134    //      Provision for lower kinematic model and upper kinematic
135    //      model consistent with ISO STEP standard.
136
137    //      Include services for characterizing these errors :
138
139    //      Include provision for
140    //      - geometric errors of motion
141    //      - thermally induced errors
142
143    //      The posFeedBackGain and the velFeedBackGain are
144    //      calculated using the connectivity of the jointCompts.
145
146    //      The basic synthesis model is the ISO standard for
147    //      kinematic modeling, which is close to the D-H model
148    //      which had   its genesis in robotics, priimarily oriented
149    //      toward a single robotic device.  Since manufacturing
```

```
150    //    equipment could consist of multiple such devices working
151    //    on a single workpiece or a set of workpieces, we extend
152    //    the ISO kinematic model, to provide for the inclusion of
153    //    kinematic models for fixtures, workpieces, and tooling.
154    //    The D-H model is also extended to include kinematic
155    //    errors of motion, the composed property of interest is
156    //    the motion of the work-point as a result of motions of
157    //    the Axis (or vice versa).  The kinematics model also
158    //    supports the model of dynamics and states.
159    {
160      attribute double Ks;
161      attribute double posFeedBackGain;
162      attribute double velFeedBackGain;
163      attribute UpperKinematicModel ukm;
164      attribute LowerKinematicModel lkm;
165      attribute CoordinateFrame placement;
166    };
167    interface Axis_Limits
168    //Limits to Motions Ranges
169    {
170      // Misc. parameters
171      attribute LinearVelocity maxVelocity;
172      attribute LinearJerk JerkLimit;
173      attribute Force maxForceLimit;
174
175      attribute Length usefulTravel;
176      attribute Length cutOffPosition;
177
178      // Following Error levels: warning, limit, violation
179      attribute Length warnLevelFollError;
180      attribute Length followingErrorViolationLim;
181      attribute Length followingErrorWarnLim;
182      attribute Length followingErrorWarnAmt;
183
184      // Overshoot Error Levels: warning, limit, violation
185      attribute Length overshootWarnLevelLimit;
186      attribute Length overshootLimit;
187      attribute Length overshootViolationLim;
188      // Amount of overshoot
189      attribute Length overshootWarnLevelAmt;
190
191      // Underreach Error Levels: warning, limit, violation
192      attribute Length underreachWarnLevelLimit;
193      attribute Length underreachLimit;
194      attribute Length underreachViolationLim;
195      // Amount of undershoot
196      attribute Length underreachtWarnLevelAmt;
197
```

```
198      // OverTravel Limits
199      attribute Length softFwdOTravelLim;
200      attribute Length softRevOTravelLim;
201      attribute Length hardFwdOTravelLim;
202      attribute Length hardRevOTravelLim;
203    };
204
205    interface Axis_Maint
206    //     Provision for data and operations that support
207    //     maintenance, e.g. health-tests, health-monitoring.
208    {
209      // Originally * pointer
210      attribute MaintHistory mh;
211    };
212    interface Axis_Operation
213    //Data concerning current operation.
214    {
215      attribute OPER_MODE operMode;
216      attribute Length holdPosition;
217      attribute LinearAcceleration requiredAcceleration;
218      attribute Force requiredForce;
219      attribute Boolean jogAxis;
220      attribute Length desiredPosition;
221      attribute Boolean hold;
222      attribute LinearVelocity desiredVelocity;
223      attribute LinearVelocity feedrate;
224      attribute double feedrateOverride;
225      attribute Boolean inPosition;
226    };
227    interface Axis_Positioning_Servo
228    {
229      void start_position_following_action();
230      void position_update_action();
231      void end_position_following_action();
232      boolean position_following_error();
233    };
234    interface Axis_Rates
235    {
236      //Specifications of travel capabilities.
237      //worst-case conditions.  But to take advantage of more
238      //capability provide a model that describes conditions
239      //when more capability is available and the corresponding
240      //values or value-functions.
241
242      attribute Length maxTravel;
243      attribute LinearVelocity maxVelocity;
244      attribute LinearAcceleration maxAcceleration;
245      attribute LinearJerk maxjerk;
```

```
246      attribute Length posErrRatioIdleStationary;
247      attribute Length posErrRatioIdleMoving;
248      attribute Length posErrRatioCutStationary;
249      attribute Length posErrRatioCutMoving;
250      attribute long repeatability;
251    };
252    interface Axis_Sensed_State
253    {
254
255      //if(!hardFwdOTravel) && if(!softFwdOTravel) &&if(!hardRevOTravel) &&
256      //    if(!softRevOTravel)
257      //then  enablingPrecondition  =  1;
258      //else  enablingPrecondition  =  0;
259      //    Concurrency: Sequential
260      Boolean getEnablingPrecondition();
261      void setEnablingPrecondition();
262
263      attribute Boolean inPosition;
264      attribute Boolean softFwdOTravel;
265      attribute Boolean hardFwdOTravel;
266      attribute Boolean softRevOTravel;
267      attribute Boolean hardRevOTravel;
268      attribute Boolean followingErrorWarn;
269      attribute Boolean followingErrorViolation;
270      attribute Boolean overShootViolation;
271      attribute Boolean enablingPrecondition;
272
273      // Addition 12/12/96
274      Axis_Position get_actual_position();
275      Axis_Velocity get_actual_velocity();
276      Axis_Accel  get_actual_acceleration();
277      Axis_Force  get_actual_force();
278    };
279    interface Axis_Setup
280    //Services perparatory to automatic cyclic operation. Data that can be supplied
281    // before arrival of current motion command.
282    {
283      // sets the reference to the axis rates for physical limits, software limits.
284      attribute AxRates physicalLimits;
285      attribute AxRates currentRates;
286      attribute AxDyn AxD;
287      attribute MOT_OBJ motionObjective;
288    };
289    interface Axis_Velocity_Servo
290    {
291      void start_velocity_following_action();
292      void velocity_update_action();
293      void end_velocity_following_action();
```

```
294      boolean velocity_following_error();
295    };
```

4.11   Control Law

```
1    interface CONTROL_LAW
2    {
3      // ATTRIBUTES
4      attribute double x;
5      attribute double xdot;
6      attribute double xdotdot;
7      attribute double output;
8      attribute double actual;
9      attribute double following_error;
10     attribute double xoffset, Ooffset, xprimeoffset;  //  new 10/96
11
12     // OPERATIONS
13     API::Status calculate_control_cmd();
14     API::Status init();   // clear time history
15   };
16
17   // PID Example
18   interface PID_TUNING // Example
19   {
20     // ATTRIBUTES
21     attribute double Kp,Ki,Kd;
22     attribute double Kx, kxdot, Kxdotdot, Kxdotprime;
23   };
24
25   // Example 1: Software Interfacce to PID Hardware Board
26   // NULL_CONTROL_AW has same api but does not cause any action
27   //interface PIDHard: NULL_CONTROL_LAW, PID_TUNING;
28   // Example 2: Software PID implementation
29   //interface PIDSoft: CONTROL_LAW, PID_TUNING;
```

4.12   Human Machine Interface

```
1    interface HMI :
2    {
3      // Presentation Methods
4      void present_error_view();
5      void present_operational_view();
6      void present_setup_view();
7      void present_maintenance_view();
8
9      // Events - to alert HMI that something has happened
10     void update_current_view();
11   };
```

12

4.13   Process Model

```
1    // Level 1
2    interface ProcessModel
3    {
4      OacVector get_user_coordinate_offsets();
5      void get_user_coordinate_offsets(OacVector offsets);
6      OacVector get_axes_coordinate_offsets();           // used by axes group
7      void get_axes_coordinate_offsets(OacVector offsets);   // set by sensor process
8      Measure  get_feedrate_override_value();           // used by axisgroup
9      void  set_feedrate_override_value(Measure feed);  // used by hmi
10     Measure  get_spindle_override_value();            // used by axisgroup
11     void  set_spinlde_override_value(Measure feed);   // used by hmi
12   };
13
```

4.14   Kinematics

```
1    // Level 1
2    // This is a basic device kinematic model
3    interface Kinematics_Model
4    {
5      AxisValues inverse_kinematics(CoordinateTransform xform);
6      CoordinateTransform forward_kinematics(AxisValues vals);
7      void set_configuration( /* TBD */ );
8      void set_tool_transform(CoordinateTransform xform);
9      CoordinateTransform get_tool_transform() ;
10     void set_base_transform(CoordinateTransform xform);
11     CoordinateTransform get_base_transform() ;
12     void set_part_coordinate_offset(CoordinateTransform xform);
13     CoordinateTransform get_part_coordinate_offset() ;
14     void set_machine_coordinate_offset(CoordinateTransform xform);
15     CoordinateTransform get_machine_coordinate_offset() ;
16   };
17
18   // Level 2
19   // This is a more sophisticated method in order to specify the kinematic ring
20
21   // Basic Kinematic definition
22   interface KinStructure
23   {
24     const UnboundedListByReference<Connection> get_connection() const;
25     void set_connection(const UnboundedListByReference<Connection> value);
26     const UnboundedListByReference<KinStructure> get_kinStructure() const;
27     void set_kinStructure(const UnboundedListByReference<KinStructure> value);
28     const CoordinateFrame get_placementFrame() const;
```

```
29      void set_placementFrame(const CoordinateFrame value);
30      const CoordinateFrame get_baseFrame() const;
31      void set_baseFrame(const CoordinateFrame value);
32    };
33    interface Connection
34    {
35     public:
36      const KinStructure * get_from() const;
37      void set_from(KinStructure *const  value);
38
39      const KinStructure * get_to() const;
40      void set_to(KinStructure *const  value);
41
42      const CoordinateFrame get_placement() const;
43      void set_placement(const CoordinateFrame value);
44    };
45
46    // FIXME: A template  would map into IDL sequence
47    typedef RWTPtrSlist<Connection> Connections;
48
49    interface KinMechanism
50    {
51      void deriveKinematicTransform();
52      const KinStructure * get_kinStructure() const;
53      void set_kinStructure(KinStructure *const  value);
54      const Connections get_connections() const;
55      void set_connections(const Connections value);
56      const Connection * get_connection() const;
57      void set_connection(Connection *const  value);
58      const KinMechanisms get_kinMechanisms() const;
59      void set_kinMechanisms(const KinMechanisms value);
60    };
61
62    // FIXME: A template  would map into IDL sequence
63    typedef RWTPtrSlist<KinMechanism> KinMechanisms;
64
```

4.15   Machine to Machine Interface

This is an outline of the final specification.

```
1     // MMSDEFS.IDL
2     // General definitions
3
4         typedef string<32> MMSIdentifier;
5     // CORBA does not seem to have an identifier type
6     // The MMSIdentifier consists of a string of characters selected from
7     //   the letters a-z, the letters A-Z, the digits 0-9, and the underscore
8     //   character. The MMSIdentifier must begin with an alphabetic character.
```

```
 9    //   The MMSidentifier is case sensitive.

10

11    // mmssverr.idl
12    // idl definitions for mms service errors
13    // see ISO 9506-2:1990 clause 7.5.5 ServiceError

14

15    #include mmsspdef.idl
16    // mms supporting definitions - to get ObjectName

17

18    #include mmspidef.idl
19    // mms program invocation definitions - to get ProgramInvocationState

20

21    #include mmsfidef.idl
22    // mms file definitions - to get FileSource

23

24        enum VMD_STATE_ERROR_CODE {
25          vmd_other,
26          vmd_state_conflict,
27          vmd_operational_conflict,
28          domain_transfer_problem,
29          state_machine_id_invalid
30        };

31

32        enum APPLICATION_REFERENCE_ERROR_CODE {
33          application_reference_other,
34          application_unreachable,
35          connection_lost,
36          application_reference_invalid,
37          context_unsupported
38        };

39

40        enum DEFINITION_ERROR_CODE {
41          definition_other,
42          object_undefined,
43          invalid_address,
44          type_unsupported,
45          type_inconsistent,
46          object_exists,
47          object_attribute_inconsistent
48        };

49

50        enum RESOURCE_ERROR_CODE {
51          resource_other,
52          memory_unavailable,
53          processor_resource_unavailable,
54          mass_storage_unavailable,
55          capability_unavailable,
56          capability_unknown
```

```
57          };
58
59          enum SERVICE_ERROR_CODE {
60            service_other,
61            primitives_out_of_sequence,
62            object_state_conflict,
63            service_reserved,
64            continuation_invalid,
65            object_constraint_conflict
66          };
67
68          enum SERVICE_PREEMPT_ERROR_CODE {
69            service_preempt_other,
70            timeout,
71            deadlock,
72            cancel
73          };
74
75          enum TIME_RESOLUTION_ERROR_CODE {
76            time_resolution_other,
77            unsupportable_time_resolution
78          };
79
80          enum ACCESS_ERROR_CODE {
81            access_other,
82            object_access_unsupported,
83            object_non_existent,
84            object_access_denied,
85            object_invalidated
86          };
87
88          enum INITIATE_ERROR_CODE {
89            initiate_other,
90            initiate_reserved_1,
91            initiate_reserved_2,
92            max_services_outstanding_calling_insufficient,
93            max_services_outstanding_called_insufficient,
94            service_CBB_insufficient,
95            nesting_level_insufficient
96          };
97
98          enum CONCLUDE_ERROR_CODE {
99            conclude_other,
100           further_communication_required
101         };
102
103         enum CANCEL_ERROR_CODE {
104           cancel_other,
```

```
105          invoke_id_unknown,
106          cancel_not_possible
107       };
108
109       enum FILE_ERROR_CODE {
110          file_other,
111          filename_ambiguous,
112          file_busy,
113          filename_syntax_error,
114          content_type_invalid,
115          position_invalid,
116          file_access_denied,
117          file_non_existent,
118          duplicate_filename,
119          insufficient_space_in_filestore
120       };
121
122       typedef long OTHERS_ERROR_CODE // *** note restriction (long)
123
124       // place holder for companion standard errors
125       enum CS_ERROR_CODE {
126       };
127
128       enum SERVICE_SPECIFIC_CHOICES {
129          obtain_file,
130          start,
131          stop,
132          resume,
133          reset,
134          deleteVariableAccess,
135          deleteNamedVariableAccess,
136          deleteNamedVariableList,
137          deleteNamedType,
138          defineEventEnrollment_Error,
139          fileRename,
140          additionalService // reserved for companion standards
141       };
142
143       union SERVICE_SPECIFIC switch( SERVICE_SPECIFIC_CHOICES ) {
144          case obtain_file :    OBTAIN_FILE_ERROR obtain_file_error;
145          case start :        START_ERROR start_error;
146          case stop :        STOP_ERROR stop_error;
147          case resume :        RESUME_ERROR resume_error;
148          case reset :        RESET_ERROR reset_error;
149          case deleteVariableAccess :
150            DELETE_VARIABLE_ACCESS_ERROR delete_variable_access_error;
151          case deleteNamedVariableAccess:
152        DELETE_NAMED_VARIABLE_ACCESS_ERROR delete_named_variable_access_error;
```

```
153        case deleteNamedVariableList :
154      DELETE_NAMED_VARIABLE_LIST_ERROR delete_named_variable_list_error;
155        case deleteNamedType :
156          DELETE_NAMED_TYPE_ERROR delete_named_type_error;
157        case defineEventEnrollment_Error :
158          DEFINE_EVENT_ENROLLMENT_ERROR define_event_enrollment_error;
159        case fileRename :    FILE_RENAME_ERROR file_rename_error;
160        case additionalService :
161          ADDITIONAL_SERVICE_ERROR additional_service_error;
162        default :    long dummy; // service specific info not present
163      };
164
165
166      typedef FileSource OBTAIN_FILE_ERROR;
167      typedef ProgramInvocationState START_ERROR;
168      typedef ProgramInvocationState STOP_ERROR;
169      typedef ProgramInvocationState RESUME_ERROR;
170      typedef ProgramInvocationState RESET_ERROR;
171      typedef unsigned long DELETE_VARIABLE_ACCESS_ERROR;
172      typedef unsigned long DELETE_NAMED_VARIABLE_ACCESS_ERROR;
173      typedef unsigned long DELETE_NAMED_VARIABLE_LIST_ERROR;
174      typedef unsigned long DELETE_NAMED_TYPE_ERROR;
175      typedef ObjectName DEFINE_EVENT_ENROLLMENT_ERROR;
176 //      enum FILE_RENAME_ERROR { source_file, destination_file };
177      typedef FileSource FILE_RENAME_ERROR;
178      typedef unsigned long ADDITIONAL_SERVICE_ERROR;
179      // The ADDITIONAL_SERVICE_ERROR really should be NULL
180      // Can I do that in IDL?
181
182      exception vmd_state_error {
183        VMD_STATE_ERROR_CODE     error_code;
184        long              additional_code; // note restriction (long)
185        string        additional_description;
186        SERVICE_SPECIFIC    service_specific;
187      };
188
189      exception application_reference_error {
190        APPLICATION_REFERENCE_ERROR_CODE     error_code;
191        long              additional_code; // note restriction (long)
192        string        additional_description;
193        SERVICE_SPECIFIC    service_specific;
194      };
195
196      exception definition_error {
197        DEFINITION_ERROR_CODE     error_code;
198        long              additional_code; // note restriction (long)
199        string        additional_description;
200        SERVICE_SPECIFIC    service_specific;
```

```
201          };
202
203          exception resource_error {
204            RESOURCE_ERROR_CODE     error_code;
205            long               additional_code; // note restriction (long)
206            string        additional_description;
207            SERVICE_SPECIFIC    service_specific;
208          };
209
210          exception service_error {
211            SERVICE_ERROR_CODE     error_code;
212            long               additional_code; // note restriction (long)
213            string        additional_description;
214            SERVICE_SPECIFIC    service_specific;
215          };
216
217          exception service_preempt_error {
218            SERVICE_PREEMPT_ERROR_CODE     error_code;
219            long               additional_code; // note restriction (long)
220            string        additional_description;
221            SERVICE_SPECIFIC    service_specific;
222          };
223
224          exception time_resolution_error {
225            TIME_RESOLUTION_ERROR_CODE     error_code;
226            long               additional_code; // note restriction (long)
227            string        additional_description;
228            SERVICE_SPECIFIC    service_specific;
229          };
230
231          exception access_error {
232            ACCESS_ERROR_CODE    error_code;
233            long               additional_code; // note restriction (long)
234            string         additional_description;
235            SERVICE_SPECIFIC    service_specific;
236          };
237
238          exception initiate_error {
239            INITIATE_ERROR_CODE    error_code;
240            long               additional_code; // note restriction (long)
241            string         additional_description;
242            SERVICE_SPECIFIC     service_specific;
243          };
244
245          exception conclude_error {
246            CONCLUDE_ERROR_CODE     error_code;
247            long               additional_code; // note restriction (long)
248            string         additional_description;
```

```
249        SERVICE_SPECIFIC    service_specific;
250      };
251
252      exception cancel_error {
253        CANCEL_ERROR_CODE    error_code;
254        long              additional_code; // note restriction (long)
255        string         additional_description;
256        SERVICE_SPECIFIC    service_specific;
257      };
258
259      exception file_error {
260        FILE_ERROR_CODE    error_code;
261        long              additional_code; // note restriction (long)
262        string         additional_description;
263        SERVICE_SPECIFIC    service_specific;
264      };
265
266      exception others_error {
267        OTHERS_ERROR_CODE    error_code;
268        long              additional_code; // note restriction (long)
269        string         additional_description;
270        SERVICE_SPECIFIC    service_specific;
271      };
272
273      exception cs_error {
274        CS_ERROR_CODE         error_code;
275        long              additional_code; // note restriction (long)
276        string         additional_description;
277        SERVICE_SPECIFIC    service_specific;
278      };
279
280   #include "mmssverr.idl"
281   // To get errorcode definitions
282
283   // VMDSupport interface
284   interface VMDSupport {
285
286   // methods - Status, UnsolicitedStatus, GetNameList,
287   //           Identify, GetCapabilityList
288
289   // Definitions for Status and UnsolicitedStatus
290
291      enum VMDLogicalStatus {
292        STATE_CHANGES_ALLOWED,
293        NO_STATE_CHANGES_ALLOWED,
294        LIMITED_SERVICES_PERMITTED,
295        SUPPORT_SERVICES_ALLOWED
296      };
```

```
297
298        enum VMDPhysicalStatus {
299          OPERATIONAL,
300          PARTIALLY_OPERATIONAL,
301          INOPERABLE,
302          NEEDS_COMMISSIONING
303        };
304
305        typedef sequence <boolean,128> VMDLocalDetail;
306
307        struct Status_Response {
308          VMDLogicalStatus    lstat;
309          VMDPhysicalStatus    pstat;
310          VMDLocalDetail    ldetail;
311        };
312
313        typedef boolean Status_Request;
314
315        void Status( in Status_Request req, out Status_Response rsp)
316          raises( resource_error, service_error, service_preempt_error,
317                   access_error, others_error);
318
319        void UnsolicitedStatus( in Status_Response rsp );
320
321    // Definitions for GetNameList
322
323        enum VMDObjectClassChoices { VMDOBJECTCLASS, VMDCSOBJECTCLASS };
324
325        enum VMDOBJECTCLASSES { namedVariable, scatteredAccess,
326                   namedVariableList, nameType, semaphore,
327                   eventCondition, eventAction, eventEnrollment,
328                   journal, domain, programInvocation,
329                   operatorStation };
330
331        union VMDExtendedObjectClass switch(VMDObjectClassChoices) {
332          case VMDOBJECTCLASS :        VMDOBJECTCLASSES oc;
333          case VMDCSOBJECTCLASS :    int dummy;
334        };
335
336        enum VMDObjectScopeChoices {
337          vmdSpecific,
338          domainSpecific,
339          aaSpecific
340        };
341
342        union VMDObjectScope switch(VMDObjectScopeChoices)
343        {
344          case vmdSpecific :    int dummy;
```

```
345          case domainSpecific:    MMSIdentifier id;
346          case aaSpecific:     int dummy;
347        };
348
349        struct GetNameList_Request {
350          VMDExtendedObjectClass     class;
351          VMDObjectScope          scope;
352          MMSIdentifier              continueAfter;
353        };
354
355        struct GetNameList_Response {
356          sequence <MMSIdentifier>    listOfIdentifier;
357          boolean               moreFollows;
358        };
359
360        void GetNameList( in GetNameList_Request req,
361                   out GetNameList_Response rsp)
362          raises( vmd_state_error, resource_error, service_error,
363              service_preempt_error, access_error, others_error );
364
365    // Definitions for Identify
366
367        typedef int Identify_Request;
368
369        struct Identify_Response {
370          string<64>     vendorName;
371          string<16>     modelName;
372          string<16>      revision;
373          sequence <MMSObjectIdentifier> listOfAbstractSyntaxes;
374        };
375
376        void Identify(in Identify_Request req, out Identify_Response rsp)
377          raises( resource_error, service_error, service_preempt_error,
378                access_error, others_error );
379    };
380
381    interface VMDSupport; // Corresponds to Remote device verification and probing
382    interface VariableAccess; // Corresponds to Polled Data Acquisition, Programmed
383                                    //  data acquistion, and parametric control
384    interface FileAccess; // No correspondence in IEC 1131-5
385    interface ResourceManagement; // Corresponds to Application Program Transfer
386    interface ProgramInvocation; // Corresponds to Program execution and IO control
387    interface OperatorCommunication; // No correspondence in IEC 1131-5
388    interface SemaphoreManagement; // Corresponds to Application Program Synch.
389    interface EventManagement; // Corresponds to Alarm notification
390    interface Journal Management; // No correspondence in IEC 1131-5
391
392    interface VMDSupport {
```

```
393        void Status(in req, out rsp);
394        void UnsolicitedStatus(in req, out rsp);
395        void GetNameList( in req, out rsp);
396        void Identify(in req, out rsp);
397        void GetCapabilityList(in req, out rsp);
398    };
399
400    interface VariableAccess {
401        void Read(in req, out rsp);
402        void Write(in req, out rsp);
403        void InformationReport(out rsp);
404        void GetVariableAccessAttributes(in req, out rsp);
405    };
406
407    interface FileAccess {
408        void FileDirectory(in req, out rsp);
409        void ObtainFile(in req, out rsp);
410        void FileOpen(in req, out rsp);
411        void FileRead(in req, out rsp);
412        void FileClose(in req, out rsp);
413        void FileRename(in req, out rsp);
414        void FileDelete(in req, out rsp);
415    };
416
417    interface ResourceManagement {
418        void ResourceDownload(in req, out rsp);
419        void ResourceUpload(in req, out rsp);
420        void RequestResourceDownload(in req, out rsp);
421        void RequestResourceUpload(in req, out rsp);
422        void ResourceLoad(in req, out rsp);
423        void ResourceStore(in req, out rsp);
424        void ResourceDelete(in req, out rsp);
425        void GetResourceAttributes(in req, out rsp);
426    };
427
428    interface ProgramInvocation {
429        void CreateProgramInvocation(in req, out rsp);
430        void DeleteProgramInvocation(in req, out rsp);
431        void Start(in req, out rsp);
432        void Stop(in req, out rsp);
433        void Resume(in req, out rsp);
434        void Reset(in req, out rsp);
435        void Kill(in req, out rsp);
436        void GetProgramInvocationAttributes(in req, out rsp);
437    };
438
439    interface OperatorCommunication {
440        void input(in req, out rsp);
```

```
441        void output(in req, out rsp);
442    };
443
444    interface SemaphoreManagement {
445        void TakeControl(in req, out rsp);
446        void RelinquishControl(in req, out rsp);
447        void ReportSemaphoreStatus(in req, out rsp);
448        void ReportSempahoreEntryStatus(in req, out rsp);
449    };
450
451    interface EventManagement {
452        void EventNotification(in req, out rsp);
453        void AcknowledgeEventNotification(in req, out rsp);
454    };
455
456    interface Journal Management {
457        void InitializeJournal(in req, out rsp);
458        void WriteJournal(in req, out rsp);
459        void ReadJournal(in req, out rsp);
460        void GetJournalStatus(in req, out rsp);
461    };
462
```

References

[Alb91]   J.S. Albus. Outline for a theory of intelligence. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(3), may/june 1991.

[COR91] Object Management Group, Framingham, MA. *Object Management Architecture Guide, Document 92.11.1*, 1991.

[DCO]   Distributed Common Object Model.
               See Web URL: http://www.microsoft.com/oledev/olemkt/oledcom/dcom95.htm.

[IEC93]   International Electrical Commission, IEC, Geneva. *Programmable controllers Part 3 Programming Languages, IEC 1131-3*, 1993.

[Inta]   International Organization for Standardization. *ISO 10303-42 Industrial Automation Systems and Integration Product Data Representation and Exchange - Part 42: Integrated Resources: Geometric and Topological Representation.*

[Intb]   International Organization for Standardization. *ISO 10303-42 Industrial Automation Systems and Integration Product Data Representation and Exchange - Part 105: Integrated Application Resources: Kinematics.*

[M.S86]   M.Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *6th International Conference on Distributed Computing Systems*, pages 198–204. IEEE Computer Society Press, May 1986.

[NGI]   Next Generation Inspection System (NGIS).
               See Web URL: http://isd.cme.nist.gov/brochure/NGIS.html.

[OMA94] Chrysler, Ford Motor Co. , and General Motors. *Requirements of Open, Modular, Architecture Controllers for Applications in the Automotive Industry*, December 1994. White Paper − Version 1.1.

[OSA96] OSACA. European Open Architecture Effort.
See Web URL: http://www.isw.uni-stuttgart.de/projekte/osaca/english/osaca.htm, 1996.

[PM93] F. M. Proctor and J.L. Michaloski. Enhanced Machine Controller Architecture Overview. Technical Report 5331, National Institute of Standards and Technology, December 1993.

[SOS94] National Center for Manufacturing Sciences. *Next Generation Controller (NGC) Specification for an Open System Architecture Standard (SOSAS)*, August 1994. Revision 2.5.